版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview
www.broadview.com.cn

MANNING

Mesos 实战

[美] Roger Ignazio 著
余何 陈秋浩 杨永帮 译

Mesos in Action

Florian Leibert 为本书作序

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

关于作者

Roger Ignazio 是一名经验丰富的系统工程师，专注研究分布式、具备容错性和伸缩性的基础架构。他对于通过更好的自动化、工具化和报告来提高生产效率极富热情。目前他是 Mesosphere 工程团队的技术指导人员，与他的妻子 Sarah 及他们的两只猫居住在美国俄勒冈州波特兰市。

Mesos实战

[美] Roger Ignazio 著

余何 陈秋浩 杨永帮 译

Mesos in Action

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

《Mesos 实战》为读者介绍 Apache Mesos 集群管理器及以应用程序为中心的基础架构概念。本书充满了有用的数据图表及实践指导，它将指引你迈出创建一个高可用的 Mesos 集群的第一步，接着在生产环境中部署应用程序，最后编写适合自己数据中心的“本地”Mesos framework（计算框架）。你将学习到如何对数千个节点进行弹性伸缩，同时通过 Linux 和 Docker 容器保证不同的进程间能实现资源隔离。你也将学习到如何使用热门主流的 framework 来部署应用程序的实践技术。

本书包含的主要内容有：搭建启动你的第一个 Mesos 集群；Mesos 的调度、资源管理及日志记录；使用 Marathon、Chronos 和 Aurora 部署容器化的应用程序；使用 Python 编写 Mesos framework。

阅读本书的读者需要熟悉数据中心管理的核心理念，也需要了解 Python 或者类似编程语言的基础知识。

Original English Language edition published by Manning Publications, USA. Copyright © 2016 by Manning Publications. Simplified Chinese-language edition copyright © 2017 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2016-6365

图书在版编目（CIP）数据

Mesos实战 / (美) 罗杰·英格纳齐奥(Roger Ignazio) 著；余何，陈秋浩，杨永帮译. —北京：电子工业出版社，2017.5
书名原文：Mesos in Action
ISBN 978-7-121-31164-2

Ⅰ. ①M… Ⅱ. ①罗… ②余… ③陈… ④杨… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2017)第060460号

策划编辑：符隆美
责任编辑：徐津平
特约编辑：顾慧芳
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：16.25　字数：364 千字
版　　次：2017 年 5 月第 1 版
印　　次：2017 年 5 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序一

世界著名的曼宁出版社（Manning）出版了不少广受欢迎的计算机丛书，如搜索领域的 *Lucene in Action*、*Elasticsearch in Action*，现在，他们又出版了这本云计算领域的 *Mesos in Action*。

Mesos 是一个开源的集群任务调度管理系统。现在随着分布式系统的广泛应用，越来越多的任务运行在集群上，而不是在单台服务器上。在 x86 PC 服务器集群上运行任务的好处是：单台服务器成本低，集群可以随着负载的增加添加服务器，水平扩展 scale out，而不是过去使用昂贵服务器的 scale up。随着集群规模的扩大，节点数越多，某个节点出现问题的概率就越大，当某个节点出现问题时，如何保证在这个节点上运行的任务能够顺利执行完成，成为一个技术难题。另外，如何管理集群，如何分发任务、监控任务执行过程等都是挑战。如果对于运行在集群上的任务，工程师还是需要在各台服务器上部署和管理，工作量将非常大，现在有些大规模集群的服务器数量已经超过万台。理想的情况是，工程师不需要关心集群里具体的每台服务器，而是把整个集群看成是一个计算、存储资源，把任务提交给集群的管理系统，由集群的管理系统去分发任务、监控任务执行，当某台服务器出现故障时，集群管理系统自动把任务派发到其他服务器上运行。这样的集群管理系统可以看作集群操作系统，甚至是数据中心操作系统。Google 在这方面做了大量的实践，在 2009 年发表的 *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale*

Machines 文章中，把整个数据中心看成一台计算机，所有的资源都由数据中心操作系统进行调度管理。

在 20 世纪 80 年代，学术界就开始了集群任务调度管理系统的研究，美国威斯康辛大学的研究人员开发了 Condor 系统，后来演进成开源的 HTCondor 系统。Google 在 1998 年成立之初，就使用 PC 服务器抓取、索引、检索全世界的网页，服务器数量巨大，他们先是开发了 WorkQueue 系统，也就是一个任务队列，工程师把需要集群运行的任务提交给这个任务队列，由任务队列把任务下发到集群的服务器，并监控任务的运行，如果服务器出现故障，就把任务重新下发到新的服务器上。后来，他们在 WorkQueue 的基础上开发了 Borg 系统，Borg 就是 Google 的集群任务调度管理系统。10 年前我在 Google 工作的时候，每天需要在集群上运行的任务，都是通过 Borg 来提交、管理的，非常方便。那时候，Google 的一个集群就已经有 2 万台服务器的规模，一个数据中心有 10 个这样的集群，20 万台服务器。我加入腾讯后，也开发了这样的系统。这种系统的资源隔离采用了容器技术（操作系统之上的资源隔离），而不是虚拟机（物理服务器之上的资源隔离），以实现更小的系统开销，更方便的管理。现在容器技术也正在被广泛使用，成为虚拟机之外的可选技术路线。

Mesos 也是一个这样的系统。Mesos 由著名的美国加州大学伯克利分校的 AMPLab 发明，AMPLab 是 Algorithms，Machines and People（算法、机器和人）实验室的缩写。AMPLab 的研究成果非常多，现在应用广泛的大数据处理框架 Spark 就是 AMPLab 的发明，AMPLab 的研究人员还包括了著名的美国科学院及工程院院士、发明了 RAID（磁盘阵列）和 RISC（简约指令架构）的 David Patterson 教授。Mesos 系统也在 Twitter、Airbnb 和苹果公司得到应用。

本书译者之一余何是 IT 专家，他曾在平安科技工作多年，有着丰富的大规模集群系统的开发、运维、管理经验，经历了多个云计算、大数据系统在金融行业的应用，在 2015 年出版了《PaaS 实现与运维管理》一书，由他作为经验丰富的实战者来翻译这本书是最合适不过的了。

相信这本书会为对大规模分布式系统、集群任务调度管理、云计算和大数据感兴趣的读者带来受益。

陈军[1]

日志易 CEO

1 陈军先生拥有18年IT及互联网研发管理经验，曾就职于Cisco、Google、腾讯和高德软件，历任高级软件工程师、专家工程师、技术总监、技术副总裁等职务，负责过Cisco路由器研发、Google数据中心系统及搜索系统研发、腾讯数据中心系统和集群任务调度系统研发、高德软件云平台系统研发及管理，对数据中心自动化运维和监控、云计算、搜索、大数据和日志分析具有丰富的经验。他拥有美国南加州大学计算机硕士学位，发明了4项计算机网络及分布式系统的美国专利。

推荐序二

余何兄是我的老朋友了，在运维领域耕耘多年，不仅运维能力强，而且善于总结乐于分享，也是 GOPS 全球运维大会金牌讲师。我和余何兄最开始是在圈子中互相关注，之后在一次论坛上偶遇，一番交流下来，一拍即合，一同走上了追求快乐运维的道路。我们的共识是“一起愉快地玩耍”，让运维变得更加轻松，让运维人员更加健康地生活。

但凡一个好的产品，都是从 0 到 1，而不是从 0 到无穷、从 0 到包罗万象，这其中的道理就是专注。专注于做好一件事，提供稳定兼容的接口，这就是一个好产品的伊始。在运维平台领域，由于其需求范围广、组织差异大，很难有那么一个产品能够满足一切，而短时间满足一切又很可能意味着 bug 多多，因此，我们应该问自己，你到底需要什么？

刚好，Mesos 就是这样一个产品：专注于做好一件事，专注于资源管理。关于其上的应用任务调度，无论是服务、批处理还是大数据，它都提供了稳定而兼容的接口，从而让用户在其上按照自己的需求，不断迭代实现，最后形成自己的产品。其在提高集群资源利用率，服务自动化部署等方面的表现，尤其令人称赞。

希望余何兄组织翻译的本书，能得到您的喜爱，为 Mesos 在中国的继续壮大添砖加瓦。为了我们共同的运维事业，一起加油！

萧田国
高效运维社区发起人开放运维联盟主席
国内第一个 DevOps Master

推荐序三

我一直在关注国内 IT 运维的发展，在不同的业务领域、企业规模下，运维的标准与规则参差不齐，真正掌握运维真知的人绝不仅意味着玩转创新技术、流行框架，更重要的是如何真正解决企业问题，如何保证落地与实现。国内第一本 PaaS 原创畅销书作者“众生的大师兄”余何有着十多年运维实战经验，经历了中国运维发展的各个阶段，在电信、金融以及物流领域都有所耕耘，今天由他领衔翻译的《Mesos 实战》同样保持了高水准，相信一定会广受欢迎。

Mesos 并没有什么吸引眼球的华丽外表，在起步阶段由于具备一定门槛，同时效能只能在具备一定规模后才能体现，因此一直没有快速地在运维领域流行起来。倒是 OpenStack 借助云计算概念、K8s 背靠谷歌这个亲爹在社区内刮起了一股大的旋风，回头再来看看近几年商业化的过度炒作，连一向低调的运维领域也产生了泡沫。回到问题的本身，为什么要使用 Mesos，我会站在和译者一样的角度考虑，我们必须考虑绕不过的环节，上层的应用，我们需要对遗留应用架构系统进行兼容吗？需要。Mesos 是一个通用性资源管理框架，能够适配各种计算类型的服务，正因为如此，向上的任务调度才稍显复杂，为了做好兼容，我们必须做更多工作。

资源管理策略 Dominant Resource Fairness(DRF) 是 Mesos 的核心，是将 Mesos 比作分布式系统 Kernel 的根本所在。Mesos 能够保证集群内所有用户都能够平等地使用集群内资源，这里的资源包括 CPU、内存、磁盘等。在通用性方面，Mesos

只负责提供资源给上层任务调度 framework，而不负责具体任务管理，于是可让各种类型计算任务使用集群资源。关于准入门槛，如果仅部署一套 Mesos，我们几乎什么也干不了，为了使用好它，我们需要不同的 Mesos framework，像 Marathon，chronos 等，在特殊场景下，甚至需要开发自己的 framework，除此之外磁盘、网络等问题也需要着重考虑。如果没有强大的意志力，初学者大多会望而却步。还好有“众生的大师兄”这样一位谆谆不倦的运维发展践行者，带着对运维未来美好发展的憧憬，坚持不懈的推行运维理念与实践，我也希望本书能够帮助 Mesos 在运维社区中快速流行与成熟，大家都能有所贡献与分享，真正解决我们企业内部遇到的运维问题。

肖力

云技术社区创始人

译者序

云计算时代，对开源产品的选择，很容易陷入一个误区，用商业化方式进行判断，选择最“热门畅销”的产品，而忽视了自身需求及组织能力，从而导致目标的偏离。2014 年，我与小伙伴们开始考察一系列平台产品并进行研究，当时我们的需求很明确，提升资源利用率与运维效率，没有其他（专注，专注，再专注）。我们的组织能力也很明确，深入理解操作系统，能够像使用黑魔法一样改变操作系统的行为，具有工程设计能力，能很快地驯服与改造各种开源产品。基于以上两点，经历了一段考察期后，我们最终选择了 Mesos。

Mesos 很难让人一下子就亲近，她绝不是让你一见钟情的那种，她没有华丽外表让你如痴如醉，也没有什么直接功能让你耍酷摆炫，你需要花很长一段时间去理解她，你需要有应用场景，有资源节约型需求，有大规模机器集群管理，有多种计算类型，之后在不断的运用中，才慢慢发现 Mesos 的奥妙之处，一步步坠入到她的爱河中。Mesos 专注于数据中心资源管理，专注于做好一件事，干净、简洁，如同操作系统内核一样，它是数据中心资源管理的 Kernel。

很多人问我 Mesos 是否可以解决运维领域的可靠性问题、资源管理问题，从此天下太平、安枕无忧，作者在书中也有一点这种态度。但很遗憾，实际运维环境是相当复杂的，这种神一样的工具几乎不可能存在。不同的管理结构有着不同的权限层级，不同的应用类型有着不同的服务级别，不同的组织环境有着不同的流程规范，

一言以蔽之，没有什么固定的工具可以解决不断变化的运维问题。

要保持运维水平的不断提升，最后发现这是一个组织行为学问题，是理念、人、流程、工具的结合体。组织的理念是什么？人的专业要求有哪些？对人的投入有哪些？组织理念文化又是如何影响着人，匹配的流程与工具又是什么？是这一切决定了最终的运维水平。精益思想对运维理念是一种启示，工程师文化强调了所需要的人才，ITIL 是运维流程上的最佳实践，工具则是连接人与流程的桥梁。Mesos 能否在企业发挥最大效应，看的不是 Mesos，而是组织自身。

余何

2016.11 于深圳软件产业基地

序

如果你想看到一个人抓狂的样子，你可以走到一个在数据中心手动配置并供应数十台服务器的人面前，然后说道："哇！持续追踪在那些机器运行了什么东西，肯定非常容易，也非常好玩。"

或者找一个身上常年带着传呼机以便响应服务器中断的人，并说："这听起来像一份毫无压力的工作呀。至少它保证你夜晚能睡个好觉。"

当然，事实上，管理服务器和其他数据中心基础架构一直以来都很困难和沉闷，给负责配置这些机器并响应机器故障的可怜男女带来了无数个不眠之夜。由于过去20年来公司越来越依赖于信息技术，经常会在每个服务器（或近些年的虚拟机）上配备一个应用程序，因此实际操作变得越来越困难。服务器数量动辄从一位数升级到两位数，有时甚至上升到三位数。

接着由Google、Facebook和Twitter等热门服务助燃的互联网呈爆炸式增长。由数以十亿计的智能手机、平板和其他设备助燃的移动互联网也立刻随后跟上。在任何特定时间里，数以百万计的用户可能同时在一个网站或APP中，而旧式计算技术无法再切入这样的世界。

在数据中心内，单一服务器的数据库（甚至所有单一服务器的服务）很快被分布式系统取代，其能以之前无法想象的容量来处理数据和流量。复杂庞大的应用程序也经常被微服务取代——把多组单一用途的服务分开管理，接着通过API进行连

接，最终构造成用户端的应用。虽然伸缩性提升了，但构建这些系统的学习曲线和管理系统复杂性也都随之提升。

Google 有个极好的方法，即在它自己的数据中心内使用一个叫 Borg 的系统来解决这个问题，表面上让大多数员工——如系统管理员和开发者——像管理一台大计算机的方式来管理数以万计的服务器。在 Borg 简化 Google 的操作几年后，开源的 Apache Mesos 项目粉墨登场并以相似的方式改变了其用户的生活。忽然之间，部署、运行和管理复杂分布式系统的过程变得非常简单。所有东西共享同样的机器组，而 Mesos 只需要轻松处理跑腿活儿——以可用的资源来匹配工作负载的需求。

最初我是作为一名 Twitter 的软件工程师来亲身体验这第一手转变的，在那里 Mesos 帮助攻克难堪的“失败之鲸”（fail whale，Twitter 服务宕机标志）并帮助 Twitter 达到伸缩性和可靠性的新高度。当 2012 年我到 Airbnb 工作时，它还只是一家建立了 4 年的创业公司，Mesos 又一次帮助我们的基础架构随着用户基数扩大成长——但是并不包括它的复杂性。Mesos 和其前景深深触动了我，于是我决定建立 Mesosphere 这个公司，致力于让 Mesos 为主流企业所用。

随着 Mesos 趋于热门和 Mesosphere 的扩张，我们把目标设为雇用最好的 Mesos 工程师和从业者。当我们看到 Roger Ingnazio 在 Pupet 实验室建立一个基于 Mesos 的持续集成平台的工作成果时，我们知道我们必须拥有它。在知名公司运行可伸缩的生产系统是非常宝贵的经验，而且自从加入 Mesosphere 后，Roger 的经验给我们基于 Mesos 的数据中心操作系统技术和我们的用户体验带来很多帮助。

Roger 的《Mesos 实战》让对 Mesos 和其技术生态系统感兴趣的人皆受益于他的经验。这本书是运行 Mesos 集群和安装你的第一个 framework（计算框架）的极佳指导，它同时也探索了更高级的主题，例如掌握强大的 Mesos framework（包括容器编排用的 Marathon 和大数据分析用的 Spark），甚至包括构建你自己的 framework。

无论你是在准备部署 Mesos，还是你已经运行了它，并想进一步提升你的知识，你都难以找到一个比 Roger 更好的导师，或者一本比《Mesos 实战》更好的书。

FLORIAN LEIBERT
Mesosphere 联合创始人、CEO

自序

Benjamin Hindman 主导的团队于 2009 年在加州大学伯克利分校创立了 Apache Mesos 这个研究项目。Ben 和他的团队想要通过允许多个应用共享一个单一的计算集群来提升数据中心的效率，就像多个应用可以在你的笔记本或工作站共享处理器、内存和硬盘驱动器一样。但他们想要在由许多服务器组成的现代数据中心上实现这个想法。经过 10000 行 C++ 代码的最初实现后，他们在 2010 年发布了一篇论文《Mesos：一个数据中心内部细粒度资源共享平台》（*Mesos:A Platform for Fine-Grained Resource Sharing in the Data Center*）。

没过多久，Ben 加入 Twitter 并使用 Mesos 来更好地扩展其基础架构，终结了 Twitter 难堪的“fail whale”时期——成名后 Twitter 的服务器处理能力跟不上用户的需求增长。虽然 Twitter 没有公开披露其庞大基础架构中的服务器数量，但根据其展示的在线资源和第一手资料，这个数量大约在每个集群有 10000 个 Mesos 节点。

2010 年 12 月，Mesos 项目进入 Apache 孵化器，作为 Apache 软件基金会的分支，使得项目获得 ASF 付出的全面支持。Apache Mesos 项目于 2013 年 6 月从孵化器中得以完善，现在已经是一个高等级的项目。

2013 年，Ben 和 Florian Leibert 及 Tobi Knaup 一起创立了 Mesosphere 公司。Mesosphere 的旗舰产品，数据中心操作系统（DCOS），通过为那些想要像 Airbnb、Apple 和 Netflix 一样轻易地使用 Mesos 来部署应用和扩展基础架构的企业提供全方

位解决方案，将开源项目成功地商业化。Mesos 持续成为开源 Mesos 项目的主要贡献者，并为开源社区提供 Mesos 安装包和工具。

我对于 Mesos 生态系统和大规模基础架构的初次涉足始于 2014 年，那时我开始想要使用 Mesos 在多个 Jenkins（热门的持续集成框架）实例之间共享资源。其时，Mesos 看起来像是留给那些已经知晓它的人去使用的，因为虽然那时有大量可用的线上资源，但是很难被找到，而且没有一个权威可信的来源。彼时也没有任何书籍涉及 Mesos。我写了几篇关于我的工作经验的博客，其他人看起来好像跟我一样：对这个项目都想了解多一些，但不知道从何入手。

2015 年 1 月，Manning 出版社找到我并问我是否有兴趣写一本关于 Mesos 的书。我之前从未写过书，这个请求一开始让我有点不知所措。但我也把它当作一个好机会来写一本工具书，因为当我刚开始使用 Mesos 时，我确实希望拥有这么一本书来指导我。幸运的是，Manning 的团队给我这份自由来实现这个想法。

希望你在使用《Mesos 实战》的过程中，能发现它是一个部署和管理 Mesos 集群和提升你基础架构整体效率的宝贵资源，而且它能够帮助你的团队或你的客户更快捷便利地部署应用到生产中。

Roger Ignazio
于美国俄勒冈州，波特兰

鸣谢

你正在阅读的是一本历经一年努力而产出的，关于 Apache Mesos 项目和生态系统的深度书籍。尽管我的名字在封面上，但还有许多为此书最终出版贡献力量的人。如果我不在这里感谢他们的话，那他们将继续匿名，无人知晓。我确定我的家人、朋友和我的妻子早已知晓在这次努力中，我对他们给予的支持怀有无比感激之情。

首先，我想要感谢 Mesos 社区。在每次交流中，无论是会议上、邮件列表上还是 IRC 上，每个人都给予了我极大帮助和善意。在撰写本书时，对 Mesos 代码库有超过 100 个贡献者，甚至有更多的志愿者在 Mesos 邮件列表和聊天室中回答问题和提供帮助。除了我要感谢有幸在会议中一起探讨和日常中一起工作的所有人，我还要感谢 Ben Hindman，Florian Leibert，Thomas Rampelberg，Dave Lester，Christian Bogeberg 和 Michael Hausenblas 他们提供的所有帮助。另外我还要感谢 Florian 为这本书写的前言。

接着，我在写作过程中改变了大约三分之二的工作方式，让写作此书仿佛不是一个有压力和花费时间的任务。在 Puppet 实验室，我要感谢 Scott Schneider，Colin Creeden，Cody Herriges，Eric Zounes 和 Alanna Brown 他们的支持。在我与 Manning 签约之前，我回想那时候当 Scott 问我是否真的认为自己能够写一整本关于 Mesos 的书。结果就是，真的可以！

很多人在幕后帮忙检查这本书的不同阶段并提供反馈，包括 Al Rahimi，Clive

Harber，John Guthrie，Luis Moux Dominguez，Mohsen Mostafar Jokar，Morgan Nelson，Nitin Gode，Odysseas Pentakalos 和 Thomas Peklak。特别鸣谢 Jerry Kuch 和 Chris Schaefer 的技术审阅及文字编辑 Sharon Wilkey 对原稿数不尽的修改。

最后，但肯定不止这些，我需要感谢我在 Manning 出版社的了不起的团队。我的编辑，Mike Stevens，帮助我从一开始“听起来像一堆胡言乱语”到获得一个正式的计划和签好的合同。开发编辑 Cynthia Kane 确保我总是提供正确数量的上下文（文字和图表），并帮助我成为一个更好的写作者和沟通者。最后感谢我的出版者 Marjan Bace，他不仅在编辑审阅时帮忙塑造这本书，并且完全给予了我自由去写这本书，这是我一开始使用 Mesos 梦想拥有的一本工具书。感谢你！

我对所有帮助我写成此书的人深深感激！若有遗漏未致谢，我在此致歉。

关于本书

《Mesos 实战》是一本在现实环境中学习和部署 Apache Mesos 的实用指南。在书中我将带你巡游整个项目——从对 Mesos 和容器的基础介绍，到满足生产发布的包含高可用和 framework（计算框架）认证的应用部署。我也将介绍热门（和开源）的 Mesos 应用的实际用法，来帮助你在 Mesos 集群里部署应用程序和计划作业。

尽管《Mesos 实战》是专门为中高级的系统管理员定制的书，它也适用于不同的读者。我写这本书时，尽量让系统管理员、DevOps 人员、应用管理员及软件工程师等类似从业者在通读全文时都能感到自在。尽管有些应用部署和软件开发的知识很吸引人，但我只会提供足够的，且非严格要求的背景资料。我会选择教你新的技巧来帮你自己的团队更聪明、而非更辛苦地工作。

路线图

如果你是一名想要部署首个 Mesos 集群的系统管理员或者 DevOps 人员，你得特别关注第 1 章到第 8 章。这些章节涵盖了你需要知道的安装和运行集群的所有事情，也涵盖了一些部署应用程序和计划作业的方法。第 10 章也能帮你了解如何编写自有的 Mesos 可用的应用程序。另外，本书分为三个部分。

第 1 部分介绍了 Apache Mesos 项目，比较了容器和虚拟机的区别，并且展示了部署 Mesos 集群的实际用例。

- 第 1 章介绍 Mesos 项目，在本章中涵盖了全书的关键术语和组件，介绍了整个 Mesos 架构，以及解释了在容器中部署应用与在传统数据中心部署的差别。
- 第 2 章基于第 1 章提供的介绍，在 Mesos 集群中运行了一个 Apache Spark 的数据处理作业。你将会看到实际运行在集群上的工作负载，以及观察该集群的行为。你也将会感受到 Mesos 是如何通过允许多个应用程序共享集群资源，来提升数据中心利用率的。

第 2 部分从各个细节回顾了 Mesos 的基础原理，包含安装、配置、高可用及监控。

- 第 3 章提供了在自己服务器上部署 Mesos 集群一应俱全的方法：无论服务器是在你自有的数据中心里还是运行在像 AWS 或者 Azure 的云上。你将会学习到如何安装和配置 ZooKeeper、Mesos 和 Docker，并且在章节结尾时就能安装和运行一个高可用集群。
- 第 4 章回顾了 Mesos 项目的基础原理，你将了解到 Mesos 提供的扩展性、容错性、高可用性及资源隔离。
- 第 5 章讲述了 Mesos 怎样处理日志记录及在问题发生时如何调试它们。在本章中，涵盖了包含 Mesos Web 接口、CLI 命令行、日志文件位置以及日志记录配置等主题。
- 第 6 章涵盖了在生产环境中运行 Mesos 的一些必要主题。它包含了监控 Mesos 和 Zookeeper 集群的信息，以及针对用户和 framework 之类的安全及访问控制。

第 3 部分介绍了 Mesos 的实际使用。

- 第 7 章介绍了开源的 Marathon framework，它允许你通过简单地指定资源的数量及想要的实例数，在集群的 Linux 和 Docker 容器中部署应用和长期运行的服务。
- 第 8 章介绍了开源的 Chronos framework，它允许你在集群上使用 ISO 8601 标准的时间戳部署计划作业。Chronos 允许作业根据时间表运行，或者允许它们依赖于其他的作业。同时 Chronos 支持任务在 Linux 及 Docker 容器中运行。
- 第 9 章介绍了开源的 Apache Aurora framework，它类似于 Marathon 和 Chronos，允许你在 Mesos 集群上部署应用和计划作业。不同的是，Aurora 开箱即用地支持多用户，以及需要匹配复杂的配置语言。
- 第 10 章讲述了 Mesos 的 API，包含了一个如何开发自有的 Mesos 应用程序的例子（使用 Python 编写）。
- 附录 A 提供了一个关于 Mesosphere DCOS（数据中心操作系统）的研究案例，一个企业级别的 Mesos 发行版本，附录 A 同时也讲述了如何使用 DCOS、Jenkins 和 Marathon 搭建持续部署流水线的全过程。

- 附录 B 提供了写本书时 Mesos 相关的知名项目列表。这些项目包含从 Mesos 应用程序、到代码绑定、到负载均衡及服务发现的工具。每个条目都伴随着一则简短的描述和一个关联到更多附加信息的在线链接。

源代码

本书中所有使用过的示例代码和配置文件都在 GitHub 上提供，位于 github.com/rji/mesos-in-action-code-samples。这些代码在本书的网站也有提供，位于 manning.com/books/mesos-in-action。

本书第 1 部分和第 2 部分大多数的代码都由配置片段组成，旨在支持和提升文本的可读性。第 3 部分的代码包含应用和数据处理作业的示例，它们是用来为你图解在自有的基础架构中该如何运行这些工作负载。

印刷约定

- 英文斜体和中文楷体用来介绍新的术语。
- `Courier` 字体用来指定代码示例和命令行。
- 代码中的持续（自动）换行用 ➥表示。

在线和社区资源

Mesos 项目有许多在线和社区资源。第一个，也许也是最重要的一个，是你能通过访问 http://mesos.apache.org/documentation/latest 来找到线上最新的文档。对 Mesos 的开发及对项目当前走向感兴趣的人，最佳的选择可能是查阅项目的问题追踪频道，位于 https://issues.apache.org/jira/browse/MESOS。

除了这两个资源外，你也能够通过下面的方式和 Mesos 社区里的人通信和交互。

- *邮件列表*——存在有几个和 Mesos 相关的邮件列表，但可能最重要的两个是 users 和 dev 邮件列表，更多的信息能在 Mesos 项目的社区页面 http://mesos.apache.org/community 上获得。
- *IRC*——类似于开发者或者使用者可以在 irc.freenode.net 的 #mesos 频道内聊天。
- *Planet Mesos*——作为 Mesos 社区成员的 RSS 聚合，Planet Mesos 包含来自项目维护者、贡献者及会议发表的文章和成果展示。这个聚合通过 http://planet.apache.org/mesos 提供服务。与此同时，Planet Mesos 的团队也运维着一个 Twitter 账号，它会在 RSS 订阅更新时自动更新 Twitter 信息，位于 https://

twitter.com/PlanetMesos。

- *Twitter*—项目官方的 Twitter 订阅位于 https://twitter.com/ApacheMesos。
- *MesosCon*—开始于 2014 年，MesosCon 是由 Mesos 社区举办的一个周年性会议。会议上由一些大型的互联网公司举行演讲，介绍他们是如何使用 Mesos 解决伸缩难题的。更多的信息，请访问 MesosCon http://events.linuxfoundation.org/events/mesoscon。

作者在线

购买《Mesos 实战》包含了由 Manning 出版社运行的私有网络论坛的免费访问服务。你可以对该书发表评论，咨询技术问题及从作者或其他用户得到帮助。要访问论坛并订阅它，你只需把浏览器指向 manning.com/books/mesos-in-action。该页面将提供各种信息，包括如何在注册后登录论坛，论坛提供哪些服务以及论坛上的行为守则。

Manning 对我们读者的承诺是提供一个场所，使得读者之间，以及读者和作者之间都能够发生有意义的对话。由于作者对在线论坛的贡献是自愿（并且没有报酬）的，所以对于作者任何指定次数的参与都是不能承诺的。我们建议你向作者提供一些挑战性的问题，以免他的兴趣流失。

只要本书在印，作者在线论坛及所有以前讨论的归档都能从出版社的网站上访问到。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31164

关于封面插图

《Mesos 实战》的封面是画作《来自克罗地亚，达尔马提亚，彼得罗沃·波耶的女孩》(Girl from Petrovo Polje，Dalmatia，Croatia)。“波耶”来源于斯拉夫语，取意“旷野”，用来形容广阔的平原。彼得罗沃波耶位于达尔马提亚，一个位于亚得里亚海岸的历史悠久的克罗地亚地区。达尔马提亚曾经是罗马帝国的一个省份，其历经哥特人、拜占庭人、威尼斯人和奥匈帝国的战事和统治。这个插图是《19 世纪中叶克罗地亚传统服装》选集的一个复制品，作者为 Nikola Arsenovic，由克罗地亚斯普利特市的人种学博物馆在 2003 年出版。博物馆的一个乐于助人的图书馆管理员帮忙取得此图，其原画存放于那个城镇中世纪中央的罗马中心——公元 304 年左右戴克里先大帝退位宫殿的遗迹。此书涵盖了克罗地亚不同地区古雅别致多姿的风格图画，以及服装和日常生活的描写。

着装礼仪和生活习惯在过去两百年已有了变化，而那时地区之间丰富的多样性也已消逝。现在已很难看得出不同大陆居民的区别，更别说不同临近村庄之间的差异了。也许是我们用文明多样性换取了更形形色色的个人生活——当然是更不同的和快节奏的科技生活。

如本书封面这般的古书图画，将两个世纪前丰富多样的地区生活重现于我们眼前，Manning 借以歌颂计算机行业的创造力和主动性。

目录

第1部分

你好，Mesos

Apache Mesos 是什么？ Mesos 与传统数据中心架构的区别是什么？容器与虚拟机的区别是什么？为什么你有可能会使用 Mesos？这些是我在第一部分要回答的问题。我们将通过对比传统数据中心与相关的新技术来学习 Apache Mesos 项目。你也将看到一个 Mesos 实践示例，它在集群上运行着一个 Apache Spark 数据处理作业。

1 初识Mesos

本章内容

- Mesos 介绍
- 比较 Mesos 与传统数据中心
- 明白为何以及何时使用 Mesos
- 用 Mesos 分布式架构进行工作

传统上说，数据中心内的物理机、虚拟机是典型的计算单元。它们被各种配置管理工具划分，随后在其上部署应用。这些机器通常被组织成集群来提供单一的服务，系统管理员管理其日常运作。集群最终会达到它请求处理的最大容量，随后更多的机器将在线加入进来处理负载。

2010 年，一个旨在解决扩容问题的项目在美国加州大学伯克利分校诞生。这个软件项目就是现在所知的 *Apache Mesos*，它在某种程度上对 CPU、内存、磁盘资源进行抽象，从而允许整个数据中心如同单台大服务器般运转。无需虚拟机和操作系统，Mesos 创造了一个单独底层的集群为应用提供所需资源。如图 1.1 所示一个简化的例子。

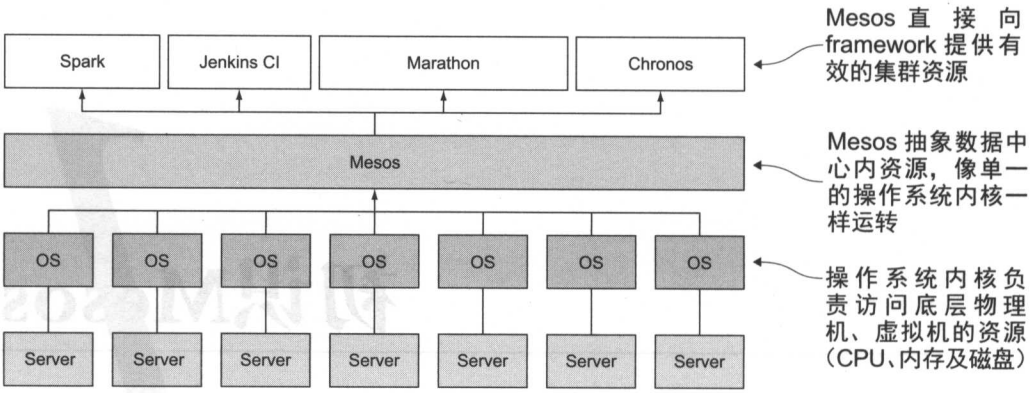


图 1.1 framework 通过 Mesos 来共享数据中心的资源

本书将介绍 Apache Mesos，一个开源的集群管理工具，它让运维和开发人员更多地关注应用本身，而不是其下的服务器资源。你将看到如何在你的环境中启动和运行 Mesos，如何共享资源和处理异常，以及可能最重要的是如何将它作为一个平台来部署应用。

1.1 遇见Mesos

Mesos 通过引入一层抽象，提供了一种像管理单台大服务器般的方法来管理整个数据中心。较关注一个应用在一个特定服务器上运行而言，Mesos 的资源隔离机制支持多租户，该功能允许多个应用运行在同一机器上，从而更加高效地利用计算资源。

为了更好地理解这个概念，你可以认为 Mesos 与当今虚拟化解决方案类似：像 hypervisor 一样抽象物理 CPU、内存、磁盘资源，之后以虚拟机形式呈现。Mesos 做相同的事情，但其将资源直接提供给应用。以另一种方式来理解 Mesos 是在多核处理器场景中：当你在笔记本上启动一个应用程序时，它运行在一个或多个处理器核中，但多数情况下运行在哪一个核上并不重要。Mesos 将这个概念应用到数据中心中。

除了提升整体资源的利用率外，Mesos 还一开始就支持分布式、高可用及容错。通过使用容器技术，如 Linux control groups (cgroups) 和 Docker，Mesos 实现了进程间隔离，允许多个应用运行在同一机器上。你也许曾搭建过三个集群，分别运行着 Memcached、Jenkins CI 和 Ruby on Rails 应用，现在你只需部署一个 Mesos 集群就可以运行所有的应用了。

在接下来的几个小节中，你将看到 Mesos 如何运作来提供所有的这些功能，以及它与传统数据中心的区别。

1.1.1 理解它如何工作

通过对资源供给、两层调度、资源隔离几个概念的整合，Mesos 提供了一种方法让整个集群像一台超级计算机一样运行任务。在进一步深入之前，让我们看一下图 1.2，该图演示了 Mesos 为应用的运行提供资源的逻辑流。在这个特殊例子里我们引用了 Apache Spark 数据处理 framework。

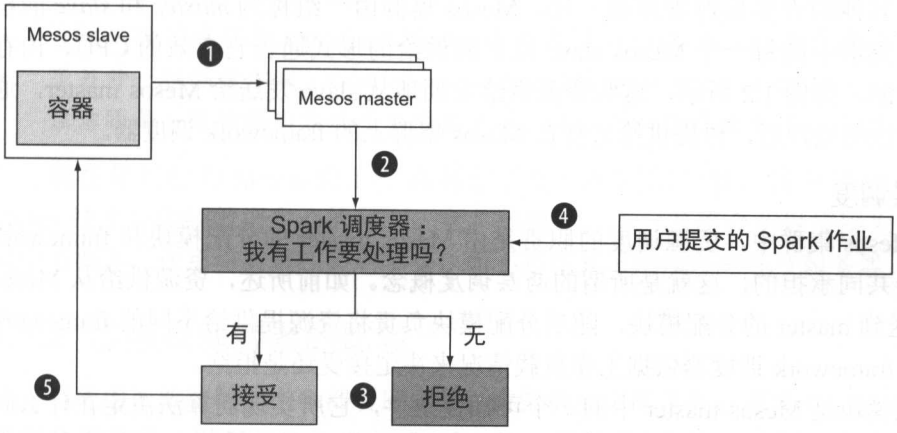


图 1.2 Mesos 将有效的 CPU、内存及磁盘突出作为资源供给 framework

让我们逐一分解：

❶ Mesos slave 将它有效的 CPU、内存及磁盘以资源供给的形式提供给 Mesos master。

❷ *Mesos master* 的分配模块（或者调度算法）决定为哪一个 *framework*（或应用程序）提供资源。

❸ 在这个特殊示例中，Spark *scheduler* 在集群中没有任何要运行的作业。它拒绝了 Mesos master 的资源供给，允许 master 将资源提供给那些可能有工作需要运行的 framework。

❹ 考虑现在有一个用户提交了 Spark 作业到集群中运行，scheduler 接受了这个作业并等待一个资源供给来满足工作负载。

❺ Spark scheduler 从 Mesos master 接受一个资源供给，随后在 Mesos *slave* 上启动一个或多个任务。这些任务在容器内运行，容器提供的隔离机制让各类任务可以运行在同一个 Mesos slave 上。

看上去很简单，不是吗？现在你已经学到了 Mesos 如何以资源供给形式来向 framework 通告资源，两层调度如何允许 framework 依据需求情况来接受或者拒绝资源供给，下面让我们进一步了解这些基础概念。

> **注意**：在未来的版本中正在努力将Mesos slave重命名为agent。因为本书采用的版本是Mesos 0.22.2，为了防止造成不必要的混淆，书中的术语均以该版本为准。更多的信息请见https://issues.apache.org/jira/browse/MESOS-1478。

资源供给

如同其他的许多集群管理器一样，Mesos集群由一组称为*master*和*slave*的机器组成。集群中的每一个Mesos slave以资源供给的形式通告它有效的CPU、内存和存储资源，如图1.2所示，这些资源供给定期地从slave发送给Mesos master，通过调度算法的处理后，再提供给运行在Mesos集群上的framework调度器。

两层调度

在Mesos集群中，资源调度的职责是由Mesos master的分配模块和framework的调度器共同承担的，这就是所谓的*两层调度*概念。如前所述，资源供给从Mesos slave发送到master的分配模块，随后分配模块负责将资源提供给不同的framework调度器，framework调度器依据工作负载情况来决定接受还是拒绝。

分配模块是Mesos master中的一个可插拔组件，它所实现的算法决定在什么时候将哪一个资源分配给哪一个framework。组件的模块化性质能让工程师为他们组织定制自己的资源共享策略。默认情况下，Mesos使用了伯克利大学开发的DRF（Dominant Resource Fairness）算法：

> 简而言之，DRF试图在所有用户间最大化它们最小资源的分配。例如，如果用户A是耗CPU型任务，用户B是耗内存型任务，DRF企图在A的CPU资源与B的内存资源之间实现平衡。在单资源场景下，DRF将回退到max-min fairness[1]算法[2]。

Mesos默认使用的DRF算法适合于绝大多数部署场景。因此你没有必要再写一份自己的分配算法，本书也不会再深入讨论DRF。如果读者对此DRF研究感兴趣的话，请阅读在线文档www.usenix.org/legacy/events/nsdi11/tech/full_papers/Ghodsi.pdf。

---

1 A.古德西，M.扎哈里呀，A.科明斯基，S.申克尔，I.斯托伊卡.“Dominant Resource Fairness:Fair Allocation of Multiple Resource Types.” NSDI, vol. 11, 2011.——注释

2 max-min fairness，算法如其名，公平地将最小资源最大化。首先保证的是公平分配，随后如果任务中有剩余资源（公平之下有剩余），则将剩余部分再均分，添加到已平均分配的资源中，如此迭代，最终出现的情况是，那些资源得不到满足的任务，它们所得到的资源是最大化的。DRF是在max-min fairness上的一次扩展，将单一资源分配扩展到多个资源上。——译者注

资源隔离

Mesos 支持多租户功能，也就是多个进程能同时在一台 Mesos slave 上运行，通过利用 Linux cgroups 及 Docker 容器技术实现了进程间隔离。framework 使用 Mesos 的容器化来执行任务。如果你对容器不熟悉，可以认为它们是 hypervisor 在一个物理主机上运行多个虚拟机的轻量级方法，不同的是容器不需要运行整个操作系统而承担额外负载。

注意：除了容器和 cgroups，Mesos 还为遵循 POSIX 标准的操作系统提供了另外一种隔离方法：`posix/cpu`、`posix/mem` 和 `posix/disk`。值得注意的是，这种隔离方法并不**隔离**资源，而是对资源的使用进行监视。

现在你已经对 Mesos 的工作机制有了更加清晰的理解，接下来你将弄清楚 Mesos 技术与传统数据中心的区别。更具体地说，下一节将介绍以应用为中心的数据中心概念，在那将着重在应用本身，而不是在其下的操作系统及服务器。

1.1.2 虚拟机和容器的比较

当考虑在传统数据中心部署应用时，虚拟机会立即浮现在脑海里。最近这些年，虚拟化提供商（以 Vmvare、OpenStack、Xen 和 KVM 为例）的服务在企业组织内部已变得非常普遍。类似于 hypervisor 抽象一台物理服务器资源给多台虚拟机共享，Mesos 也提供了一层抽象，尽管在不同级别实现。资源将直接分配给应用本身，依次被容器消费。

为了说明这一点，请考虑图 1.3，它对基础架构资源的各层进行了比较，其上部署了 4 个应用。

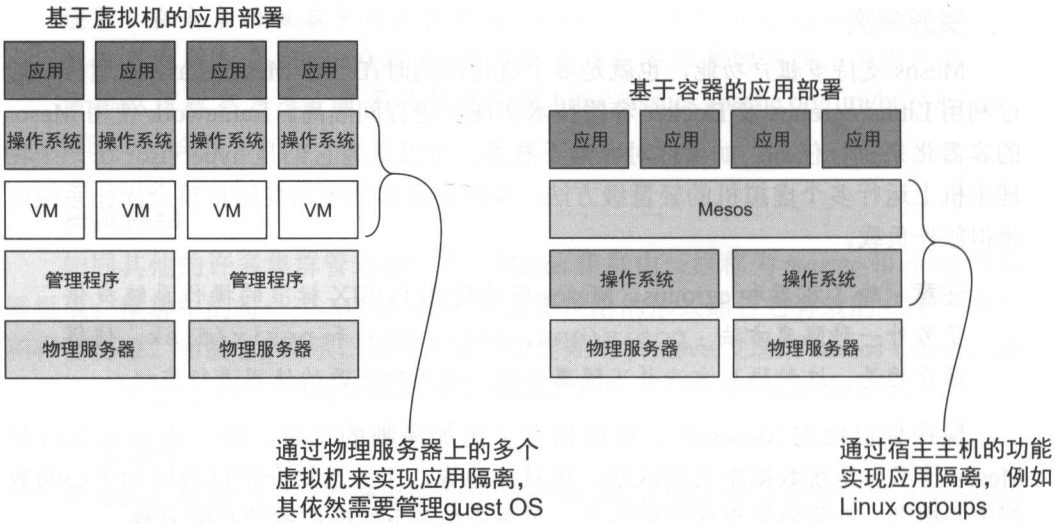


图 1.3 基于虚拟机和基于容器的应用部署对比

虚拟机

当想到基于虚拟机部署应用时，应考虑运维操作系统的开销：安装软件、应用安全更新、管理用户权限、识别与修复配置信息变化，这些步骤还将继续。当你更关心应用部署本身时，要知道在整个操作系统之上运行应用到底有什么附加的好处？何况操作系统本身也是有开销的，它消耗了 CPU、内存、磁盘资源。在应用系统大量扩容的场景下，这将变得非常浪费资源。Mesos 依据以应用为中心的方法来管理数据中心，采用轻量级的容器，能极大地简化基础架构资源栈及应用部署。

容器

如之前所见，Mesos 使用容器技术实现进程间的资源隔离。在 Mesos 的环境中，最重要的两个资源隔离方法分别是构建在 Linux 内核之上的 *cgroups* 和 Docker 容器。

2007 年左右，Linux 内核 2.6.24 版本开始支持 cgroups，它让进程可以运行在沙箱内与其他进程隔离。在 Mesos 环境中，cgroups 对运行态进程进行资源限制，确保它们不会影响同系统上的其他进程。在使用 cgroups 之前，进程所依赖的包文件、库文件（特定版本的 Python，可运行的 C++ 编译器等）必须已经安装在操作系统上。如果你的工作负载、包文件、库文件及工具集都是标准化的，或者它们彼此不易产生冲突，这也不会是一个问题。图 1.4 示范了如何使用 Docker 技术来克服这些冲突类的问题，如何以一种更好的隔离性方式来运行应用与负载工作。

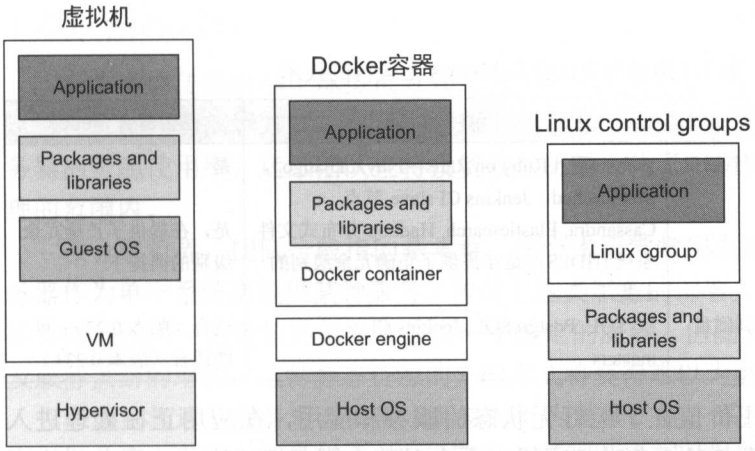


图 1.4 虚拟机、Docker 容器及 Linux cgroups 之间的比较

通过使用包括 cgroups、namespaces 的 Linux 内核底层原语，Docker 可以像虚拟机般构建和部署容器。应用程序及其所依赖的全部文件都被打包在容器内并部署到了操作系统上。它使用了运输行业的概念——工业标准集装箱，通过遵循标准来部署应用。这种新的软件单元交付方式较虚拟机部署更加轻便，近些年来越来越流行。

你并不需要了解所有的实现细节，以及使用 Mesos 构建与部署容器的错综复杂信息。如果你想了解更多的信息，请参考以下在线资源：

- Linux control groups: www.kernel.org/doc/documentation/cgroup-v1/cgroups.txt。
- Docker: https://docs.docker.com。

1.1.3 知道何时及为何使用 Mesos

运行大规模应用不再是只有大企业可以做到的了，人员不多的创业公司也能够轻松创建吸引百万用户的 APP。重新构建应用和数据中心不是一件简单的事情，但是在资源栈中某些组件是非常适合运行到 Mesos 上的。通过使用这些技术和将它们（以及它们的负载）迁移到 Mesos 集群上，你将更加容易地进行资源扩容及让数据中心更具效率。

注意：本书涵盖的是 Mesos 0.22.2 版本，它提供了一个环境运行无状态、分布式应用。Mesos 从 0.23 版本开始，将开始支持持久化资源，包括有状态的 framework，更多的信息请参看 https://issues.apache.org/jira/browse/MESOS-1554。

举例说明，考虑无状态、分布式及有状态的相关技术适合与不适合在 Mesos 上

运行的情况如表 1.1 所示。

表 1.1 适合与不适合在 Mesos 上运行的技术

服务类型	示 例	是否适合 Mesos
无状态——不需要持久化数据到磁盘	Web 应用（Ruby on Rails，Play，Django），Memcached、Jenkins CI slave 节点	是
便捷的分布式	Cassandra, Elasticsearch, Hadoop 分布式文件系统 (HDFS) | 是在提供了正确冗余级别的前提下	是，在提供了正确冗余级别的前提下
有状态——需要持久化数据到磁盘	MySQL, PostgreSQL, Jenkins CI masters	适合（版本 0.22）；可能适合（版本 0.23+）

实现 Mesos 的真正价值在于运行无状态的服务和应用，在应用正在处理进入的负载时，集群中的节点在任意时刻离线也不会对整个服务造成影响，服务或许可以运行一个作业来将这个结果报告给其他系统。如前所述，这类应用包括 Ruby on Rails、Jenkins CI slave 节点。

在 Mesos 上以 framework 形式运行分布式数据库（例如 Cassandra 和 Elasticsearch）和分布式文件系统（例如 Hadoop 分布式文件系统）已取得了不错进展。但是其可行性是建立在正确的冗余级别上的。虽然某些分布式数据库和文件系统具备数据复制和容错功能，但是如果整个 Mesos 集群出现异常（例如自然灾害、冗余电路、制冷系统异常，或者人为失误），你的数据仍将无法被抢救回来。在真实的世界中，你应该对风险进行评估，权衡在 Mesos 集群上部署持久化数据服务的利弊。

如前所述，Mesos 适合运行无状态、分布式服务。在本书写作之时，需将数据持久化到磁盘中的有状态应用并不适合于在 Mesos 之上运行。虽然可以这样做，但并不建议某些数据库服务在 Mesos 上运行，例如 MySQL、PostgreSQL。如果你需要持久化数据，推荐使用传统的数据库集群方式，部署在 Mesos 集群之外。

1.2 为什么我们要重新思考数据中心

在数据中心中部署应用一般会涉及一台或者多台服务器资源。虚拟化技术的引入并逐渐变成主流，方便我们在一台服务器上运行多个虚拟机，从而能够更好地使用物理资源。但这种方式运行应用意味着每一个虚拟机上将跑一个操作系统，这将造成更多的资源消耗，以及带来额外的维护成本。

本节将阐述我们需要重新思考如何管理数据中心的两个主要原因：资源静态分配的管理开销，以及将重心放在应用而不是基础设施的需求。

1.2.1 资源划分

当你考虑传统的基于虚拟机方式部署应用和集群静态分配时，你马上会意识到这是一种低效的部署方式，在运维管理上也是非常繁重的。通过最大化数据中心服务器的资源使用率，运维团队也将投资收益比最大化了，保证了总成本在尽可能合理的范围内。

在计算机界，团队一般指的是*集群*，这个术语常常用来表述一组共同工作的服务器作为单一系统来提供某种服务。传统上这类服务的部署方式是以节点为中心的，即你必须使用一定数量的机器来提供一个给定的服务。但随着基础架构资源的扩展及服务供给的增长，这种静态资源的分配方式将变得难以为继。

考虑到当这些服务资源的需求翻倍时，为继续扩容，系统管理员需要分配新的虚拟机并将它们加入到集群中。运维团队很可能提前意识到将来需要更多额外资源，于是将集群的能力直接扩容到之前的三倍。尽管你很好地完成了服务的扩容工作，但现在集群中的某些机器是空闲的，等待着被使用。如果任意一集群中的虚拟机突然异常，它可能需要立即恢复在线来保证服务的总容量，如图 1.5 所示。

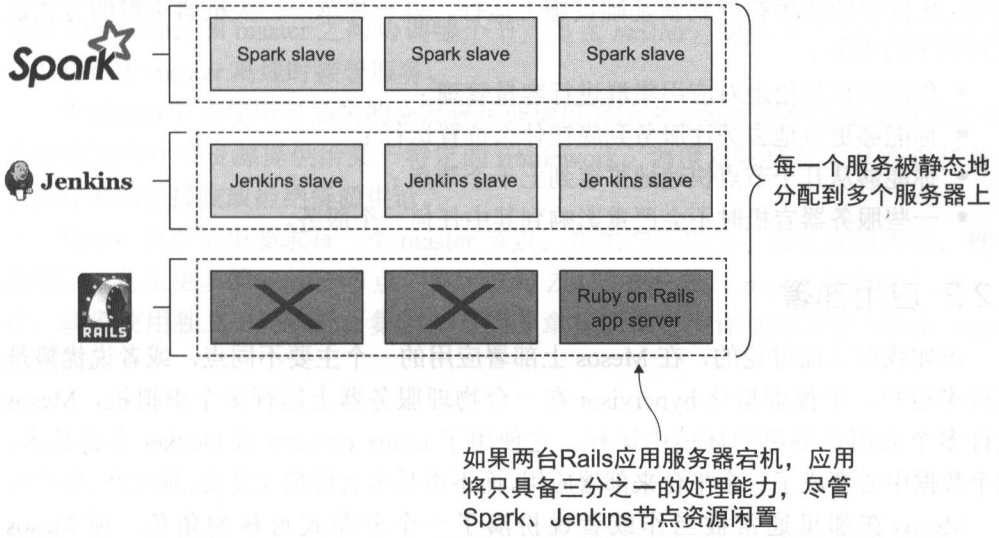


图 1.5 在数据中心中资源静态分配的三个应用

现在考虑通过 Mesos 来解决如上扩容场景的问题，如图 1.6 所示，你能看到使用了数据中心里同样的服务器，但这次是着重在应用上，而不再是虚拟机。应用可以在任意拥有有效资源的机器上运行。如果你需要扩容，你只需要将机器添加到 Mesos 集群中，而不是将机器添加到多个应用集群中。如果某个单一的 Mesos 节点离线了，对 Mesos 集群中的任何服务都不会产生影响。

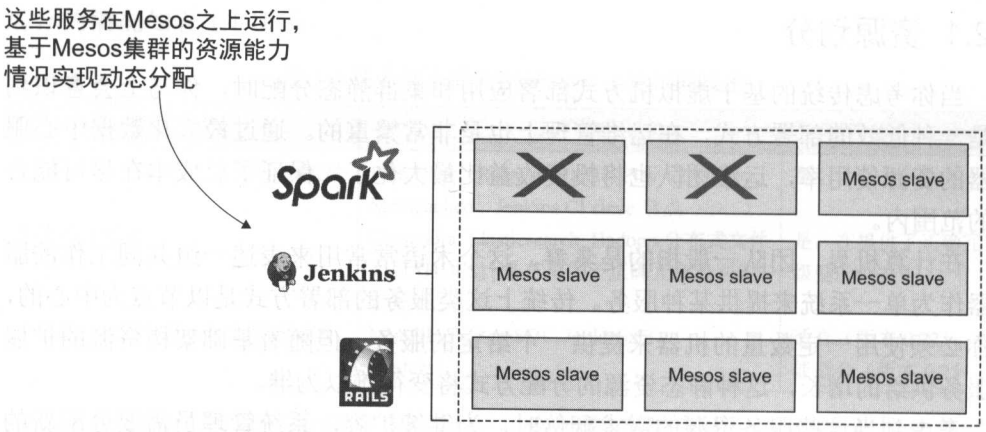


图 1.6 在 Mesos 集群中运行的三个应用

考虑一下在成百上千台服务器时的差异，你不再是去试着猜测需要多少台服务器，以及如何将它们分配到静态的应用集群中，而是允许这些服务按照它们的需求动态地请求计算、内存、存储资源。你将新的机器添加到 Mesos 集群中完成扩容，Mesos 集群中的应用扩展到新基础设施上运行。以一种单一、大机器集群的方式运维有如下的优势：

- 你能够很轻松地对应用集群进行容量管理；
- 你能够更少地去关注服务到底在什么位置运行；
- 你能够从几个节点快速地扩容到上千个节点；
- 一些服务器宕机时不会严重影响到其中任何一个服务。

1.2.2 应用部署

正如我们上面讨论的，在 Mesos 上部署应用的一个主要不同点，或者说优势是支持多租户。不像虚拟化 hypervisor 在一台物理服务器上运行多个虚拟机，Mesos 允许多个应用在隔离的环境中运行。它使用了 Linux *cgroups* 或 Docker 容器技术。整个数据中心变成了一个平台来部署应用，而不再是多套环境（开发、预生产、生产）。

Mesos 在那里通常被当作或者说扮演了一个分布式内核的角色，而 Mesos framework 帮助用户来执行长运行态服务，以及调度任务，类似于 init、Corn 系统这种周期性执行任务。你将在后面学到这些 framework（Marathon、Chronos 和 Aurora）的更多知识，以及如何在其上部署应用。

考虑一下到目前为止我所描述的功能：Mesos 提供了非常易用的容错能力。当一台服务器离线时，Mesos 集群会自动在另一处服务器上启动之前异常失败的服务，而不是让系统管理员手工地卷入进来。只有在大量机器同时离线的时候系统管理员

才需要介入，因为那可能是一个大问题。正因如此，在正确的位置放置和资源冗余充足的情况下，日常维护可以在任何时候进行。

1.3 Mesos分布式架构

为了提供规模化服务，Mesos 提供了一套分布式、容错性架构来完成资源的细粒度分配。这套架构包括三个组件：*master*、*slave* 及运行在其上的应用本身（通常称为 *framework*）。Mesos 依赖于 Apache ZooKeeper，一个分布式的数据存储系统，专用于集群内的协调同步 leader 投票选举，以及 Mesos master、slave 和 framework 间的 leader 发现。

在图 1.7，你能够看见这些架构组件如何在一起工作，从而提供一个稳定的平台来部署应用。在接下来的章节中我将分解开来一一加以阐述。

1.3.1 masters

Mesos master 的职责是管理集群中在每台机器上运行的 Mesos slave 守护进程。通过 ZooKeeper 和 master 之间协调哪个节点是主 *master*，哪些节点作为备用存在，它们将在主 master 离线时接管服务。

主 master 节点使用可插拔的分配模块或调度算法来分发资源供给至各种调度器，从而决定将什么资源提供给某一特定的 framework。调度器依据其上是否有任务需要执行来决定接收或拒绝资源供给。

Mesos 集群至少要求有一个 master 节点。在生产环境为了保证高可用性，推荐采用三个甚至更多的 master 节点。你可以将 ZooKeeper 在与 master 相同的机器上运行，或者使用独立 ZooKeeper 集群。在第 3 章我们将更加详细地讨论 Mesos master 的部署。

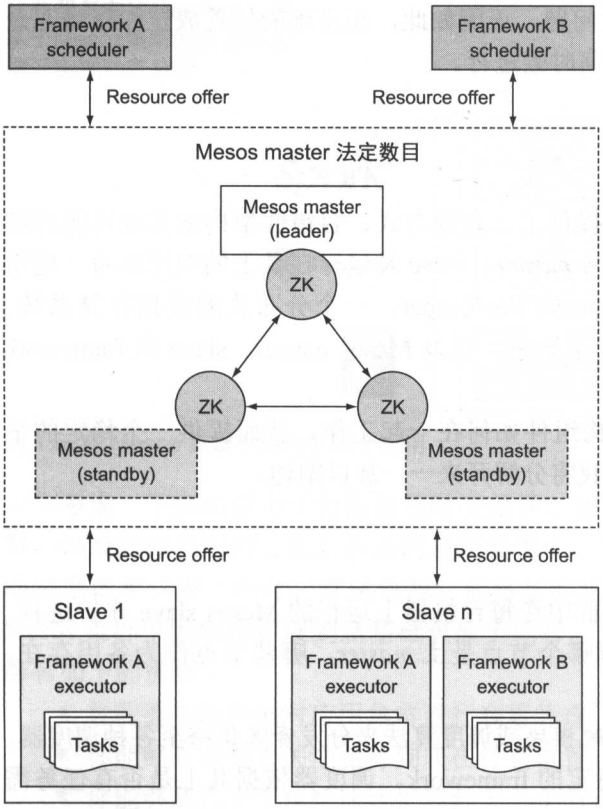


图 1.7 Mesos 架构图，包括一个或多个 master、slave 和 framework

1.3.2 slaves

在集群中负责执行 framework 任务的服务器被称为 Mesos *slave* 节点，它们访问 ZooKeeper 来确定主 master 节点，将 CPU、内存、存储资源以资源供给的形式宣告给主 master。当调度器从主 master 接收资源供给后，在 slave 节点上启动一个或多个执行器，执行器负责运行 framework 的任务。

Mesos slave 也能够基于属性与资源进行配置，从而允许它们定制特定环境。属性配置是键值对形式，可以包含类似于节点所在机房位置信息。资源配置可以替代 Mesos 自动探测发现 slave 节点的有效资源，并由用户指定具体的 CPU、内存、磁盘资源信息。属性配置与资源配置的示例信息如下：

```
--attributes='datacenter:pdx1;rack:1-1;os:rhel7'
--resources='cpu:24;mem:24576;disk:409600'
```

在属性配置中对 Mesos 节点的数据中心、节点位置、操作系统进行了说明，在

资源配置中用户指定了该节点提供的CPU、内存和磁盘资源。在日常维护中，这些信息对保证应用在线运行不受影响特别有用。使用这些配置信息，数据中心的运维人员可以在整个机柜甚至整排机柜离线日常维护时不影响用户。第4章在Mesos slave配置章节将对此进行详细讨论。

1.3.3 frameworks

如你前面所了解到的，*framework*是表示Mesos应用的术语，它负责在集群上调度与执行任务。framework由两个组件组成：调度器与执行器。

提示：本书写作时存在的framework清单请见附录B。

调度器

调度器是典型的长运行态服务，负责与Mesos Master连接，接收或拒绝资源供给。Mesos将调度的职责委派给了framework，而不是试着由自己调度所有的任务执行。调度器基于当下是否有任务需要运行来决定是否接受或拒绝资源供给。调度器通过与ZooKeeper通信来探测主master的存在，之后将其自己注册到master中。

执行器

执行器是在Mesos slave上启动的一个进程，负责运行framework的任务。在本书写作之时，Mesos内建的执行器允许framework执行shell脚本、Docker容器等。Mesos支持多种编程语言执行器，新的执行器可以与framework绑定在一起，当任务需要它时由Mesos slave从framework获取。

如你所看到的，Mesos提供了一个分布式、高可用的架构，master负责整个集群的调度工作，slave将有效资源通知调度器，并在集群中执行任务。

1.4 小结

在这一章中，你了解了Apache Mesos这个项目、它的架构，以及它是如何解决数据中心伸缩性问题的，它又是如何让集群变得简单。你也学习到了在应用部署上Mesos与传统数据中心的区别，为什么以应用为中心的方法能够让资源更加有效。我们也讨论了在什么地方（不在什么地方）使用Mesos来应对负载，你在哪儿可以获得帮助，获得更多你需要的信息。这里有如下一些事项请谨记：

- Mesos从底层系统抽象化CPU、内存、磁盘资源，将多个服务器展现成一个大机器。
- Mesos slaves以资源供给的形式宣告它有效的CPU、内存、磁盘资源。

- Mesos framework 由两个主要组件组成：调度器和执行器。
- 容器使用轻量级方法实现了进程间的资源隔离。

在下一章节，我将带着你通过运行一个真实例子来了解 Mesos 如何让资源更具有效率，你如何在自己的数据中心内将应用运行到 Mesos 生态系统上。

2 使用Mesos管理数据中心资源

本章内容

- 通过真实示例介绍 Mesos
- 对独立和通用集群进行对比
- 在 Mesos 集群上启动 Spark job
- 探索 framework 与 Mesos 间的交互

前一章介绍了 Mesos 项目，它是如何工作的，它与传统数据中心架构有什么区别。这一章将通过一个真实场景来探讨 Mesos 的益处：演示多个应用间如何采用 Mesos 来共享资源，并将用 Apache Spark 作为范例，它是一个流行的数据处理 framework。

如果你对 Spark 并不熟悉，请不必担心：接下来的场景中将使用 Spark 来演示 Mesos 是如何在多个应用间分发工作负载和实现资源共享的。我们用 Spark 作为案例来学习在通用的 Mesos 集群上进行资源共享和工作负载调度，了解通用模式与数据中心内的静态资源分配有什么区别。我们将对 Mesos 和 Spark 的 Web 界面做一个简单介绍，也许你将在这个过程中对 Spark 也有所了解吧，谁知道呢？接下来就让我们开始吧。

2.1 Spark简要介绍

引用项目主页的原话，“Apache Spark是一个快速、通用的大规模数据处理引擎”，它与另外一个流行项目Hadoop一同活跃在大数据领域，常用于数据科学分析。在很多案例中，Spark执行任务无论是在内存中还是在磁盘上，比Hadoop的MapReduce更加快速与更具高效。

Spark同时提供了各种流行编程语言的API，包括Python、Scala和Java，除了支持类MapReduce的批处理外，同时支持流式处理、交互查询、机器学习库。

Spark历史简介

2009年，Matei Zaharia在加州大学伯克利分校AMPLab实验室开始了Spark的开发，这个组织同时支持着Mesos的开发。事实上，Matei也是Mesos的联合创造人。

2010年Spark开源后，被捐赠给了Apache软件基金会，2013年在Apache内进行孵化，到了2014年升级为顶级项目。

2013年Matei作为联合创始人成立了Databricks公司，在Spark成功之后寻求将其商业化的机会，帮助客户们解决大数据方面的问题。Databricks目前仍然是Spark开源项目的最大贡献者。

在最基础的层面上，Spark需要集群管理器来分配工作负载，并访问一个Hadoop兼容的数据源。一般情况下，Spark支持如下几种类型的集群管理器：

- Spark独立版；
- Mesos；
- Hadoop YARN；
- 仿分布式（在本地的笔记本或者工作站上运行）。

虽然可以将Spark在本地运行，使用笔记本或工作站中的多核CPU，但这常常仅作为开发环境使用：CPU核的数量限制了执行器的数量，当你建立一个生产环境Spark集群时，有两个选择：部署一个独立版的静态分区的Spark集群到提前准备好的一批服务器上，或者使用一个类似于Mesos或YARN的集群管理器来运行Spark任务。

为了更好地说明如Mesos这样的通用集群器，我们将在2.1.1节中以Spark独立版集群来对比Mesos。

2.1.1 独立集群上的Spark

在图2.1中，你看到Spark driver连接到了一个集群管理器——Spark master，它

轮流将任务分发到各个工作节点。

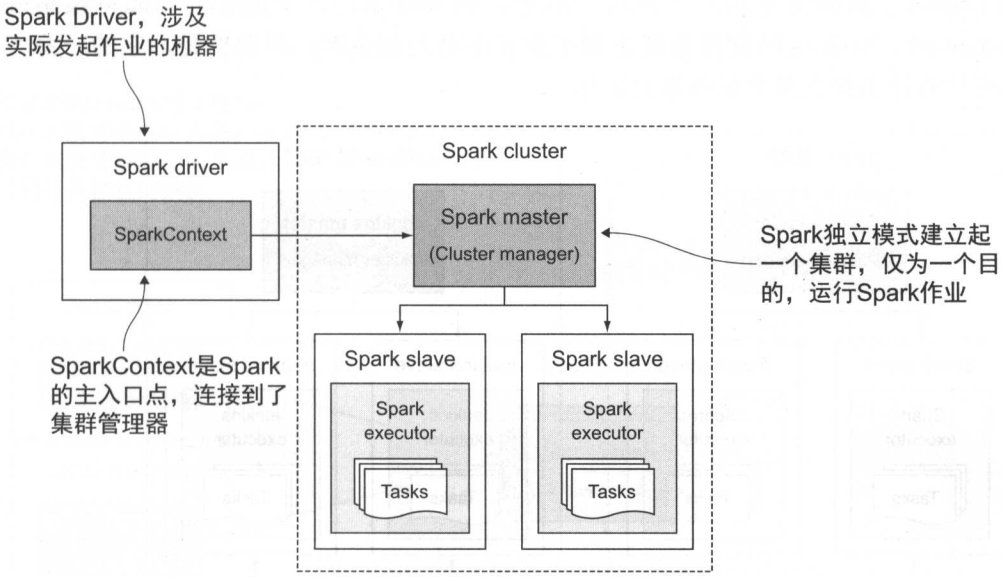


图 2.1 独立模式 Spark 的组件与架构

在图 2.1 中，*Spark Driver* 是实际发起 Spark 作业的机器，*SparkContext* 是 Spark 的主入口点，它负责连接到集群管理器，并在集群中运行任务，同时它还负责创建 Spark 的分布式数据集。如你所见，在 Spark 集群中的两个工作节点只为仅有的一个目的：它们是专门运行 Spark 任务的机器，仅此而已。

我们在第 1 章学到，Mesos 提供了一种优秀方法来在一个集群上运行多个应用，在一个工作节点上启动多个任务。你能使用 Mesos 实现跨多个应用的资源共享，而不是启动一个或者多个静态分区的 Spark 集群。让我们看看在 Mesos 上运行 Spark 是一个怎样的情形。

2.1.2 Mesos 上的 Spark

虽然启动 Spark 来使用一个独立集群并不是什么问题，但考虑一下多个团队需要他们自己的 Spark 集群，或者考虑更深一层，在数据中心内要有多个静态的分区集群。

如果你在物理机器上部署这些静态集群，显然你投入了一定数量的成本在工作负载上，而这些负载是没有办法实现资源共享的。同样的，如果你在 IaaS 云端启动一个静态的分区集群，如亚马逊 AWS 云上，你很可能因为这些云端机器的空置而浪费金钱。不管你是在本地机房还是在云端，细粒度的资源共享可以帮助你提升资源利用率，从而提升整个数据中心的效率。

为了举例说明这点，让我们看一下图2.2，你有两个独立集群服务于两个应用：Spark（数据处理的例子在这一点上）和Jenkins，一个流行的开源持续集成framework。Jenkins的使用在这个例子中并不是特别重要，重要的是，它的一些其他应用程序需要在多个服务器上运行。

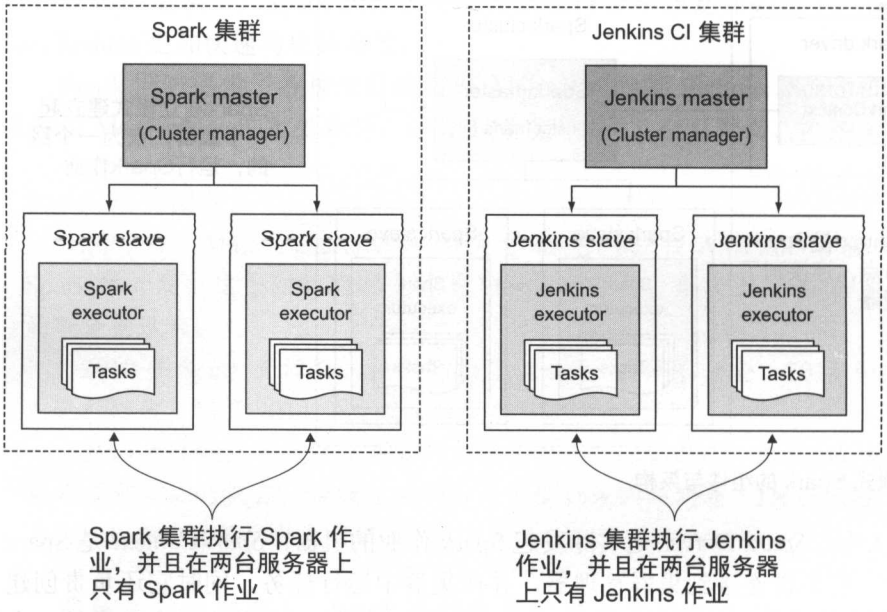


图 2.2 形象化两个静态的分区或者固定的集群

如图2.2所示，你有两个静态的分区集群，一个是Spark，另一个则是Jenkins，每一个集群包括它自己的集群管理器（Spark master和Jenkins master）和两个工作节点来启动或构建任务。你也能够清晰地看到静态分区（或称为资源储藏室，如果你愿意的话）中两台服务器出现异常，这样就没有任何办法在两个集群间共享计算机资源。但很可能这两个服务都不会同时将资源使用到100%的使用率，如果Spark集群在工作负载50%时就发现资源不足，而Jenkins集群只要50%的资源即满足负载需求，那么Spark可以通过从两台服务器变成三台服务器资源而受益。

现在，让我们考虑在一个类似于Mesos的通用集群管理器上运行这些应用，从而实现细粒度的资源共享，如图2.3所示，通过Mesos使用Linux cgroups技术将每一个framework的执行器进行了隔离，你可以在一个Mesos slave上共享计算资源并运行多个工作负载。因此，在大规模机器的现代数据中心内，跨机器间的资源共享将大幅提升资源利用率。

现在你已经花一些时间明白了Spark怎样使用Mesos作为其集群管理器，为什

么使用类似于 Mesos 的通用集群管理器能通过实现资源共享来提升效能。让我们看看 Spark 是如何在 Mesos 集群上运行的，在我们进入到第 3 章介绍 Mesos 的安装与配置之前，这将让我们更好地明白 Mesos 是如何运行任务的。

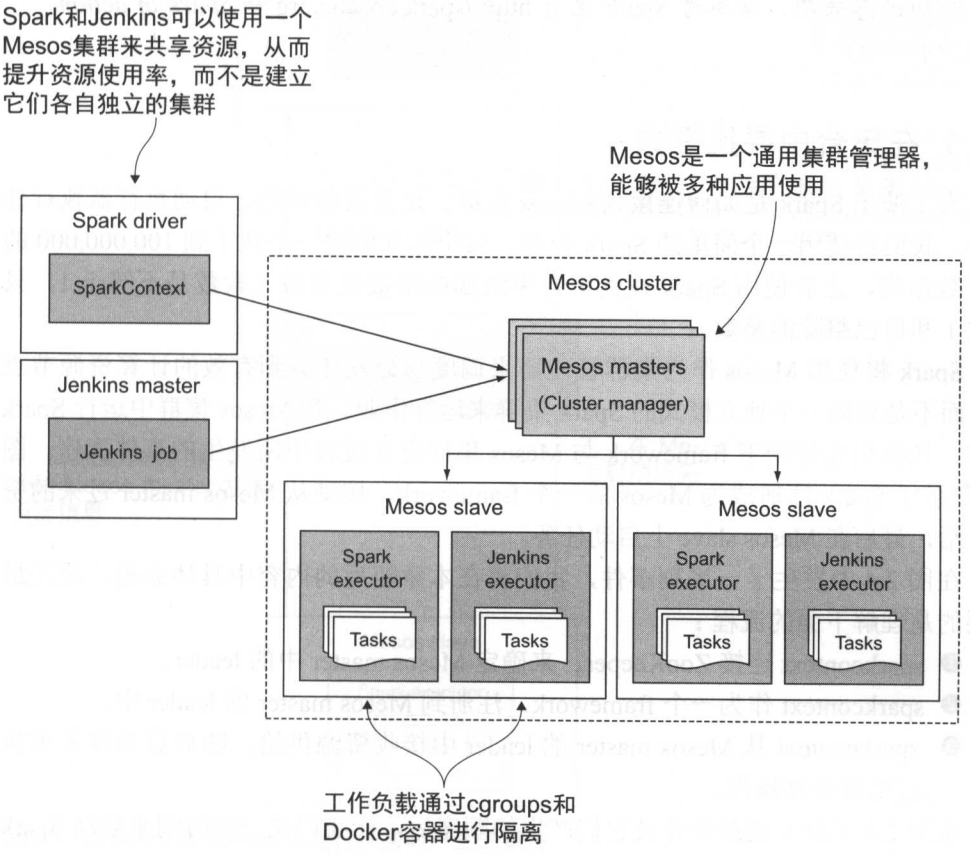


图 2.3 Mesos 为两个应用管理集群资源

2.2 在Mesos上运行Spark job

本章之前讨论的独立模式 Spark 集群的架构，很可能与你期望的其他分布式系统一样：一个主调度器和一个或多个工作节点。图 2.3 示例说明了你如何使用 Mesos 来避免将你的数据中心静态划分为多个集群，你的工作负载需要一个单一的通用集群，而不是各自进行计算资源的申请。接下来让我们通过一个例子进行 Mesos 实践，来说明 Mesos 如何为类似于 Spark 的 framework 进行任务分发负载。

注意：本节是关于在 Mesos 上下文内运行的 Spark，并不是 Spark 的入门知识。尽管我们展示了如何在 Mesos 上运行 Spark 作业，但你在此节并没有学到 Spark 在现实世界中如何使用数据处理作业。如果你对 Spark 的更多知识感兴趣，请参考 Spark 主页 http://spark.apache.org 和 *Spark in action* 一书。

2.2.1 在集合中寻找素数

为了演示 Spark 是如何连接到 Mesos 集群、接受资源供给、启动执行器执行任务的，我们将使用一个简单的 Spark 作业。这个作业创建一个从 1 到 100,000,000 的整型数据集，之后使用 Spark 找到集合中的哪些整数是素数（素数是不等于 1，只能被 1 和自己整除的整数）。

Spark 将使用 Mesos 作为集群管理器来调度与分发任务到有效的计算资源节点上，而不是启动一个独立模式的 Spark 集群来运行作业。在 Mesos 集群中运行 Spark 之前，让我们先讨论下 framework 与 Mesos 集群交互过程中所发生的事件次序。图 2.4 展示了 Spark 注册成为 Mesos 的一个 framework，接受从 Mesos master 过来的资源供给，最后在 Mesos slave 上启动任务。

在图 2.4 中发生了一系列事件，我们将在本章随后的内容中具体分析，现在最重要的是理解下面的流程：

❶ sparkcontext 连接 ZooKeeper，来确定 Mesos master 中的 leader。

❷ sparkcontext 作为一个 framework，注册到 Mesos master 的 leader 中。

❸ sparkcontext 从 Mesos master 的 leader 中接收资源供给，随后启动任务来执行数据处理操作。

从图 2.4 了解了这些事件及它们产生的顺序后，让我们花一些时间来启动 Spark 作业，观察集群中真实的输出内容。在我们启动一个 Mesos 集群并运行后（我们将在第 3 章介绍），开始安装 Spark、运行相关的示例。

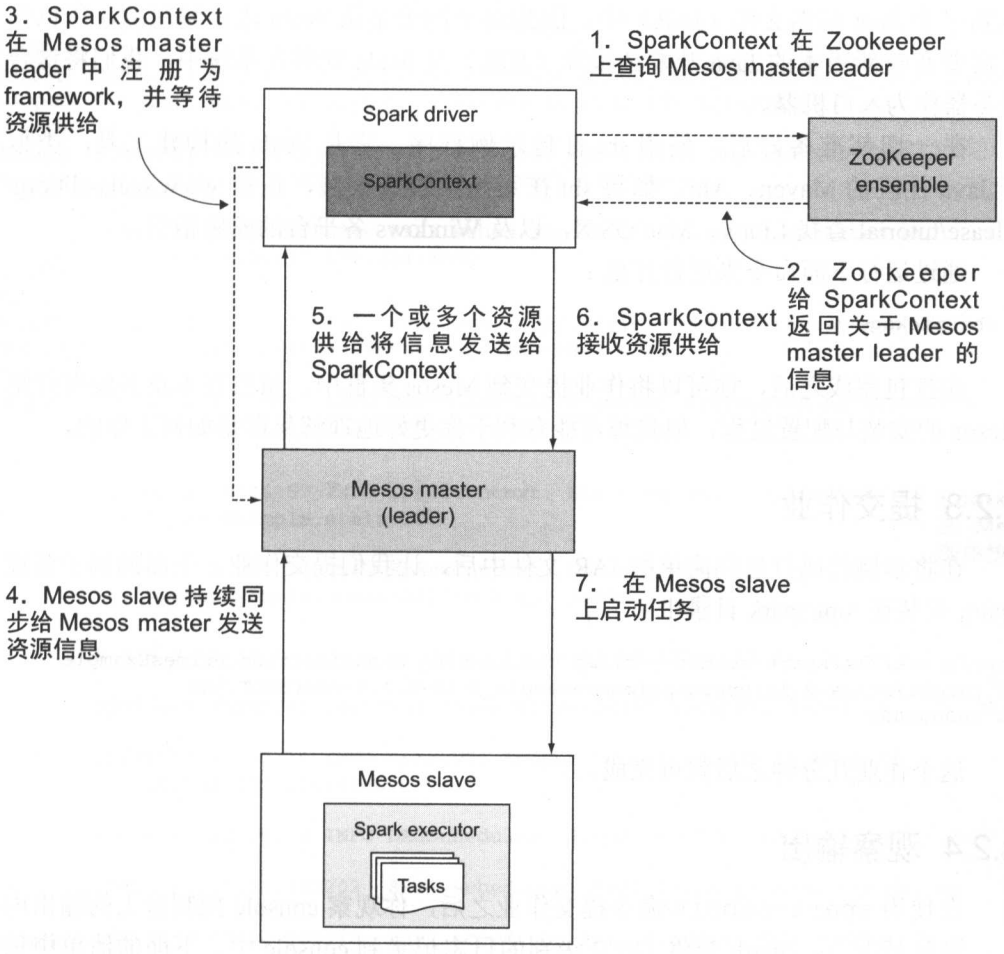


图 2.4 当 Spark 在 Mesos 上运行任务时产生的事件

提示：在 Mesos 上安装 Spark 的指引请参考 http://spark.apache.org/docs/latest/running-on-mesos.html。

2.2.2 获取与打包代码

Spark job 的示例代码作为本书的增补材料在 GitHub 和 manning.com 均有效，最简单的获取示例代码的方法是从 Git 仓库中克隆的：

```
$ git clone https://github.com/rji/mesos-in-action-code-samples
$ cd mesos-in-action-code-samples/chapter02/spark-primes-example
```

接下来，你需要使用 spark-submit 命令行工具将 Spark 作业及其依赖的内容打

包到一个 Java 归档文件（JAR）中，因为这个例子是用 Scala 语言编写的，所以你还需要有较新版本的 Java 开发环境库（JDK）及 Scala 安装在系统中。我们将这台服务器作为入门机器。

在一切都准备好后，使用 sbt 打包示例程序，它是 Scala 的构建工具，类似于 Java 社区的 Maven、Ant。如果 sbt 在系统中没有安装，请在 www.scala-sbt.org/release/tutorial 查找 Linux、Mac OS X，以及 Windows 各平台的安装指引。

通过运行下面命令来进行打包：

```
$ sbt package
```

在打包完成之后，你可以将作业提交到 Mesos 集群中，虽然在本章并没有介绍 Mesos 的安装与配置过程，但我想可能有利于你更好地理解集群是如何工作的。

2.2.3 提交作业

在将示例代码打包到简单的 JAR 文件中后，让我们提交作业。下面的例子假设 Spark 安装在 /opt/spark 目录中：

```
/opt/spark/bin/spark-submit --class com.manning.mesosinaction.PrimesExample
➥ target/scala-2.10/spark-primes-example_2.10-0.1.0-SNAPSHOT.jar
➥ 100000000
```

这个作业几分钟之后就可完成。

2.2.4 观察输出

在使用 `spark-submit` 命令提交作业之后，你观察 console 控制台上的输出内容，默认情况下，Spark 会将 `INFO` 级别的日志记录到 console 中。下面的清单中包括了日志中的一些重要消息。我将要解释它们在 Mesos 环境中的含义。

清单 2.1 Spark 作业在 Mesos 集群上运行的输出

```
15/04/12 22:35:56 INFO Utils: Successfully started service 'sparkDriver'
    on port 45957.

15/04/12 22:35:56 INFO Utils: Successfully started service
    'HTTP file server' on port 49444.

15/04/12 22:35:56 INFO SparkUI: Started SparkUI at
    http://10.132.171.224:4040
```

```
I0412 22:35:57.401646  8991 sched.cpp:157] Version: 0.22.2

2015-04-12 22:35:57,415:8901(0x7f8ed93eb700):ZOO_INFO@check_events@1703:
    initiated connection to server [10.132.171.224:2181]

I0412 22:35:57.418431  8993 detector.cpp:452] A new leading master
    (UPID=master@10.132.171.224:5050) is detected

I0412 22:35:57.418504  8993 sched.cpp:254] New master detected at
    master@10.132.171.224:5050

I0412 22:35:57.420454  8993 sched.cpp:448] Framework registered with
    20150412-214000-3769336842-5050-2832-0005

15/04/12 22:35:57 INFO MesosSchedulerBackend: Registered as framework ID
    20150412-214000-3769336842-5050-2832-0005

15/04/12 22:35:57 INFO SparkContext: Starting job: collect at
    PrimesExample.scala:22

15/04/12 22:39:34 INFO SparkContext: Job finished: collect at
    PrimesExample.scala:22, took 217.099354417 s

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79

99999839 99999847 99999931 99999941 99999959 99999971 99999989

15/04/12 22:40:10 INFO SparkUI: Stopped Spark web UI at http://
    10.132.171.224:4040

15/04/12 22:40:10 INFO DAGScheduler: Stopping DAGScheduler

I0412 22:40:10.902202  8931 sched.cpp:1589] Asked to stop the driver

I0412 22:40:10.902323  8997 sched.cpp:831] Stopping framework
    '20150412-214000-3769336842-5050-2832-0005'

15/04/12 22:40:10 INFO MesosSchedulerBackend: driver.run() returned
    with code DRIVER_STOPPED

15/04/12 22:40:11 INFO SparkContext: Successfully stopped SparkContext
```

SparkContext 在 ZooKeeper 上查询 Mesos master

SparkContext 在 Mesos 中将其自己注册为一个 framework

Spark driver 在 Mesos 集群中执行任务

为保证简洁，省略部分输出

SparkContext 停止它的调度器

Spark framework 从 Mesos master 中注销

尽管在 Spark 日志中还记录了很多事件内容，实际上在一般情况下你可能并不需要 *INFO* 级别日志，在日志清单中被选中的行很好地说明了 Spark 是如何使用 Mesos 来处理它的工作负载的，至少在 Mesos 上启动 Spark 作业时看到了发生事件的次序，如同我们在图 2.4 中所见到的一样。

现在你已经了解了真实场景下的事件次序，下面让我们用更多的方法来观察日志输出及在 Mesos 集群上运行的 framework 和任务的状态。

2.3 进一步探索

刚刚在 Mesos 集群上提交任务，运行了一个 Spark 作业，并观察控制台 console 上的输出后，现在可能是来介绍背后发生种种的好时机了。在这一节中，你将快速地浏览一下 Mesos 和 Spark 的 Web 界面，并能够在界面上实时地观察集群中任务运行的情况。我们将在第 3 章开始介绍 Mesos 的安装与配置，下面几节的内容将作为本书第 2 部分主题的引子来论述。

2.3.1 Mesos UI

Mesos master 提供了一个 Web 界面来查看集群状态和任务执行情况，此 Web 接口提供集群的相关信息，包括以下内容：

- 所有任务的概况，以及它们的状态；
- 注册的 framework，以及它们关联的任务；
- Mesos slave，它们的资源情况，以及当前执行的任务；
- 资源供给情况（那些被拒绝或者还未被接受的资源供给）。

在图 2.5 中，你能够看到在 Spark 素数示例中，framework 被注册到了集群中，当前消耗了 6 核 CPU，2.6GB 内存，并运行了多个任务。到目前为止，你不必担心需要理解 Web 界面上的每一个功能，为了让你熟悉 Mesos 有效功能，你只需要观察集群如何响应运行在其上的 Spark 作业。在第 5 章，我们将回到 Web 界面，并进行更深入的讨论。

点击这些任务的 sandbox 链接，允许你进一步看到这个 Mesos 沙盒任务的文件和日志输出内容，甚至有每一个单独任务的工作目录。在图 2.6 中，sandbox 内包含了 Spark 作业的 JAR 文件及这些文件在 console 控制台上 stdout 和 stderr（标准输出、错误输出）中捕获的输出。

2.3.2 Spark UI

除了 Mesos 提供的 Web 界面外，Spark 会启动它自己的 Web 界面来负责监控 Spark 作业的运行进展。你在清单 2.1 Spark 作业的输出中可以看到这个信息。虽然没有必要因为一些通用的 Mesos 集群功能而去访问这个 Web 界面，但它还是提供了一个漂亮的、与集群无关的 Spark 作业进展界面。

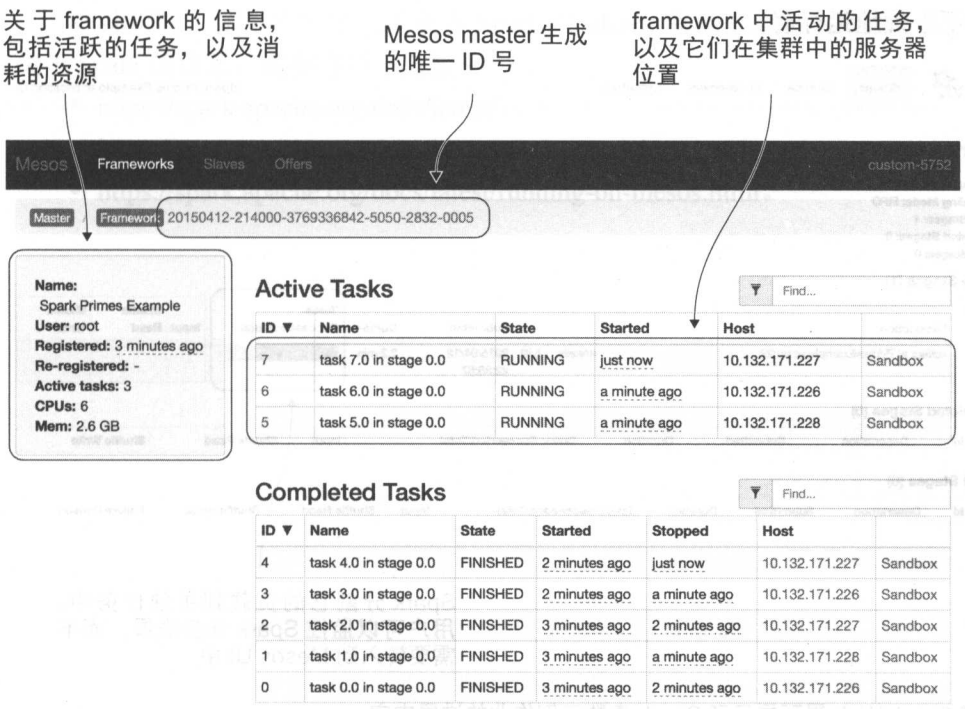


图 2.5 Spark 素数示例 framework 正在消耗集群资源以及运行任务

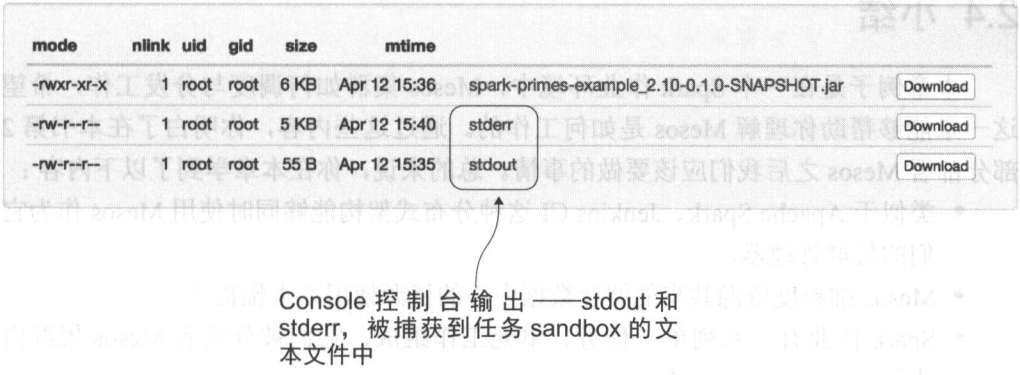


图 2.6 Mesos sandbox 中看到的一个 Spark 任务的相关文件

在图 2.7 中，你能够看到任务的当前执行阶段，“collect at PrimesExample.scala:22”正在集群上运行，并完成了 5/8 的任务量。

不管作业是运行在 Mesos 上还是在其他集群管理器上，这些任务将以类似方式分布执行。不同的是，Mesos 允许你运行多种 framework，它们的任务以隔离的方

式互不干扰地运行着。

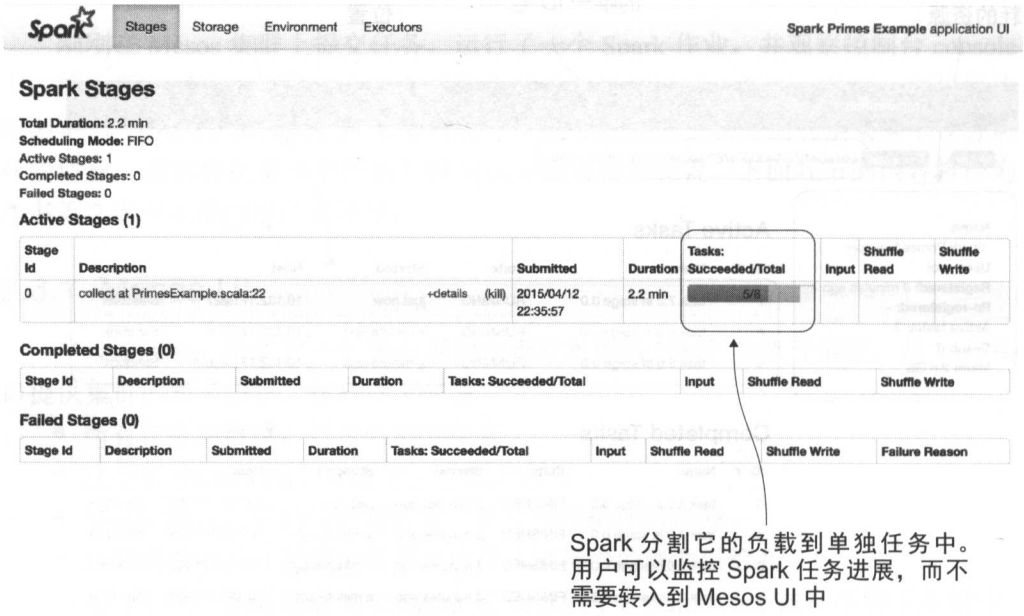


图 2.7 Spark Web 界面展示了 Spark 素数示例作业的进展内容

2.4 小结

本章例子是在一个 Spark 作业环境中，Mesos 集群如何调度与分发工作，希望这一章能够帮助你理解 Mesos 是如何工作的。通过这些内容，你明白了在本书第 2 部分部署 Mesos 之后我们应该要做的事情。总的来说，你在本章学到了以下内容：

- 类似于 Apache Spark、Jenkins CI 这种分布式架构能够同时使用 Mesos 作为它们的集群管理器。
- Mesos 细粒度资源共享能够让数据中心的资源使用率大幅提升。
- Spark 作业有一系列单一任务、单元工作组成，它们被分发到 Mesos 集群内执行。
- Mesos Web 界面提供了当前集群状态的预览。

在本书的第 2 部分中，我们将重新回顾第 1、2 章中解释的每一个概念，包括资源隔离、容错，并深入到日志、调试、slave 资源与属性等。如果你愿意，你可以从本书的增补材料（在 GitHub 与 manning.com 均有效）中下载 Spark 素数的例子，并在你的 Mesos 集群中测试它，第 3 章将包括安装与配置内容。

在此同时，如果你需要关于更多 Spark 在 Mesos 集群上运行的信息，或者更多关于 Spark 的信息，请参考以下链接：

- https://spark.apache.org/docs/latest/。
- https://spark.apache.org/docs/latest/spark-standalone.html。
- https://spark.apache.org/docs/latest/running-on-mesos.html。

第2部分

Mesos核心

在第 2 部分，我提供了在生产环境部署 Mesos 集群的基础知识。你会学习到 Mesos、ZooKeeper 和 Docker 的安装配置过程，同时也会了解到 Mesos 的高可用架构、监控配置，还有它的访问控制。

3 安装Mesos

本章内容

- 开发和生产集群的部署事项
- Mesos，ZooKeeper 和 Docker 的安装配置
- 不停机升级 Mesos

无论你是通过用配置管理工具（例如 Puppet、Chef，或者 Ansible），或者是用 SSH、Fabric 在远程系统中执行脚本和命令，了解用公共安装包来部署 Mesos 和源码编译 Mesos 都是很重要的，这样可以让你定制自己的部署，或者建立自己的安装包。

这一章会带你认识 Mesos 主备节点的安装配置，学习到 Apache ZooKeeper 的集群协调知识，还有了解 Docker 是怎样启动容器的。你还能了解到生产环境部署中的 Mesos 和 ZooKeeper 的高可用架构，也可以了解为了开发需求而在单节点安装的所有东西。

Puppet 的配置管理

一个好的配置管理策略对于一个运行良好的数据中心是至关重要的。本章运用到的很多指令——包括安装配置 Mesos、ZooKeeper 和 Docker——都是可以通过 Puppet 来执行的，它是一个开源的配置管理工具（本书在写的时候，它就是配置管理领域里面的几个工具之中最出名的一个）。下面三个特殊的 Puppet 模块能够自动化配置和维护 Mesos 集群。

- https://forge.puppetlabs.com/deric/mesos。
- https://forge.puppetlabs.com/deric/zookeeper。
- https://forge.puppetlabs.com/garethr/docker。

由于 Puppet 的用法已经在它的官方文档或者其他书籍中被很好地介绍了，这里就不再冗余阐述了。可以在大多数的 Linux 发行版本或者 puppetlabs.com 里面下载到 Puppet 的安装包。

无论你是否使用配置管理工具来部署 Mesos 集群，阅读本章对于你理解这几种不同的组件之间的依赖、交互协调都是一个不错的选择。

3.1 部署Mesos

在探究一项新的技术，部署一个新的系统的时候，无论是在开发、测试、生产环境，你都要尽力地学习掌握了解它。这样，当有些事不可避免地出错的时候，你就准备好应对了。下面几个小节会教会你几个封装 Mesos 集群的组件。你也可以学习到在开发或者生产上部署集群需要考虑的情形。

3.1.1 Mesos 集群组件

让我们再来了解一下这几个不同的组件之间是怎样相互协调的。无论你是准备在开发或者生产环境上使用 Mesos，一个集群的部署需要包含下面一些（或者全部）的组件：

- 必须的——一个或者多个 *Mesos master* 节点（如果 master 节点的个数大于 1，一定得是一个奇数）。
- 必须的——一个或者多个 *Mesos slave* 节点（总的来说，一个集群中的 slave 节点越多，效果越好）。
- 可选的——一个或者多个机器组装的 *Zookeeper*。如果想让 Mesos 处于高可用的状态，那么它是必须的（ZooKeeper 节点个数大于 1 的话，那也必须是奇数个的）。

- 可选的——*Mesos slave* 上面跑的 *Docker Engine*。

根据你自己的需求和目的，对于在开发和生产的不同环境，你可能需要考虑几个额外的信息。下面几个小节就是阐述这个主题的。

3.1.2 开发环境的注意事项

当以开发为目的去部署配置 Mesos 的时候，在单一节点部署所有的组件是挺合理的事情，选择简单的部署而不是高可用的架构。在开发环境，你需要按照下面的顺序安装、配置、部署如下的组件：

1. 单实例的 ZooKeeper。注意这个是可选的，仅当你的 framework 需要用到 ZooKeeper 做协调和维持状态的时候用到（高可用的架构也需要用到）；
2. Mesos master 服务；
3. Mesos slave 服务；
4. Docker 引擎（可选）。

当你读到后面安装配置 Mesos 小节的时候，对于在单一机器部署所有的组件就再普通不过了，因为这种情况太多了，所以 Mesosphere 上面有一个团队专门为此开发了一个叫作 Playa Mesos 的项目。

Playa Mesos 的介绍

在团队成员之间创建可持续利用开发环境的一个比较热门的方法就是使用 Vagrant（www.vagrantup.com）。Playa Mesos 是一个由 Mesosphere 创建和维护的开发环境，可以在单一虚拟机上面提供 Mesos 集群的工具，还可以用来做开发 framework 的实验，运行应用（通过 Marathon），跑调度任务（通过 Chronos）。这个工具还包含了已经提前配置安装好的一个单一实例的 ZooKeeper 和 Docker 引擎。

> **提示**：Playa Mesos 的项目可以在 GitHub 中找到 https://github.com/mesosphere/playa-mesos。

假设你已经安装好了 Vagrant（这个需要像 VirtualBox、VMware Fusion 或者 VMware Workstation 这样的虚拟化软件），那么就很容易在开发环境运行 Playa Mesos，如下：

```
$ git clone https://github.com/mesosphere/playa-mesos
$ cd playa-mesos
$ vagrant up --provision
```

在生产环境中我们也可以使用上面的方法，但是为了实现高可用的架构，也需要考虑一些额外的情形，现在我们就来看看这些具体的情形。

3.1.3 生产环境的注意事项

这一小节包含了部署一个 Mesos 集群的很多练习。然而，值得注意的是，这个小节并不是 Mesos 在生产环境跑的一个指引。它的目的是帮你在本书的这个阶段做一些集群配置上的决定。第 6 章会提供在生产环境中运行 Mesos 的更多细节，包括日志记录、监控，还有访问控制。

生产部署概览

对于生产环境，你至少需要三个 Mesos master 和三个 ZooKeeper 组成的集群。对于开发目的，你可以用单一的 Mesos master 和单实例的 ZooKeeper，但是你得注意，系统就没有冗余性了。

在图 3.1 中，你可以看到三个用 ZooKeeper 做协调的 Mesos master，通过 Mesos maser leader 沟通的不同的 Mesos slaves（这个例子上面运行的是 Docker）。

尽管 ZooKeeper 是作为一个和 Mesos master 分开部署的单独的服务，但是你还是会选择一个简单一点的部署将它和 Mesos master 部署在同一个主机上面的。不要担心记不住下面的图，我会在你学习安装配置这个集群的路上不断重复地阐述。

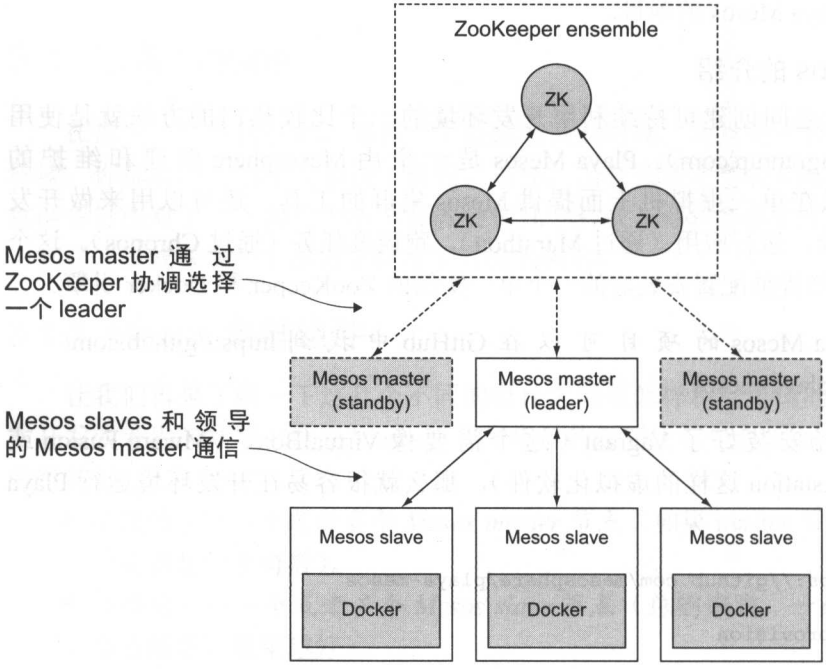


图 3.1 Mesos 集群部署组件

几乎是不言而喻的，在专用的硬件或者虚拟化设备或者云服务提供商的设备上，

部署这些服务的时候，一定要考虑到所有硬件层面上的冗余。如果你的服务器是运行在物理的数据中心上面，你的 Mesos master 和 ZooKeeper 所在的服务器应该放在不同的机架上面，并且连接到多个配电单元，诸如此类的。如果你是在虚拟化设备或者云环境上运行，则必须制定一些策略保证虚拟机在不同的虚拟化设备上或不同的可用区域里面运行。

考虑到所有的集群协调工作都是在 Mesos master 和 ZooKeeper 集群里面发生的，所以必须保持单点错误达到最小的限度。如果你有多个数据中心，或者有灾备中心，那么在网络延时足够低的情况下，也要考虑使用它们。

Mesos master

如果你计划将 Mesos 集群在不同的数据中心上面运行，确保每个数据中心和 master 之间的网络低延时性是非常必要的。否则，如果在 `registry_store_timeout`（默认 5 秒）这个时间内，主 master 不能向备份的 master 写进注册信息，那么 Mesos 的注册更新就会失败。如果需要增加这个限时的话，你可以考虑增加 `registry_fetch_timeout`（默认 1 分钟）的时间。根据你的环境而定，你可能需要在每个数据中心分别运行一套 Mesos 集群，在数据中心之间用负载均衡访问替换它们。

ZooKeeper 集群

考虑到在 Mesos masters、slaves 和 framework 中都是使用 ZooKeeper 作协调的，因此它的高可用性不用说也是需要的。一个 ZooKeeper 集群，也称为一个组合体，需要在集群内维护一个*法定数目* (*quorum*)，或者叫多数票。你可以忍受的宕机数目取决于你的环境和你与用户的服务级别协议，创建一个可以忍受 F 个节点宕机的环境，你需要部署 $(2 \times F + 1)$ 个节点，如表 3.1 所示：

表 3.1 不同 quorum 对应的 ZooKeeper 节点数

ZooKeeper 集群大小（节点数目）	法定数目	可容忍的失效机器数
1	1	0
3	2	1
5	3	2
$2 \times F + 1$	$F + 1$	F

由于 ZooKeeper 集群需要一个大多数的投票对集群做决定，通常情况下会部署单数个节点。通常来讲，一般是要求在生产环境部署 5 个 ZooKeeper 节点。这样可以使得一些节点在维护期间，集群还可以忍受一些不可预知的事故。

注意：在生产环境部署 ZooKeeper 之前，请花点时间阅读 ZooKeeper 管理

员指引，在 http://zookeeper.apache.org/doc/current/zookeeperAdmin.pdf 可以找到。

在和 Mesos master 同一个主机上安装 ZooKeeper 并不是一个要求，但它确实是简化了部署，也是一个可以接受的方案。如果你计划部署一个为其他软件服务的 ZooKeeper 集群，我推荐你将它和 Mesos 的 ZooKeeper 集群分开，单独部署，让 Mesos 的 ZooKeeper 集群可以专心为 Mesos 集群服务。这一章的后面，也是在 Meoss master 上安装 ZooKeeper 以减少复杂度。

就像我之前所说的，这一节所列的东西不是一张最佳练习的列表，而是一个对于部署 ZooKeeper 和 Mesos master 的节点数与位置的指引。第 6 章会讨论一些额外的生产环境部署需要考虑的事情，包括日志、监控和访问控制。

3.2 安装Mesos和ZooKeeper

Linux 和 Mac OS X，还有其他一些类 UNIX 的操作系统上是支持 Mesos 安装的，你有两个选择来安装 Mesos 和 ZooKeeper：

- 使用操作系统的打包工具；
- 通过源代码编译来安装二进制文件。

在这一小节，你会学到在两个最热门的 Linux 发行版本里面安装 Mesos，就是红帽企业版 Linux(REHL)/centos 7 和 Ubuntul 14.04 LTS（也叫 Trusty）。

注意：记住在一些云的部署环境里，你还是有责任管理好安装、配置和服务的健康状态的。一些云提供商，类似于亚马逊 AWS，提供一些工具让你创建自动化配备基础架构模板的准备。因为这些工具是云服务商专有的而不是 Mesos 自带的，我们就没必要阐述了，我们更偏向于说明安装过程中一些操作系统层面上的配置。你可以咨询云提供商，查看文档，运用不同的方法使得你可以自动化部署 Mesos 集群。

3.2.1 使用安装包部署

Mesosphere 为在生产环境上常见的几种不同的 Linux 发行版本提供了 Mesos 安装包，本书在写的时候，包括以下操作系统的类型：

- RHEL / CentOS 6 和 7；
- Ubuntu 12.04 到 14.04；
- Debian 7（也称为 Wheezy）。

下面例子的安装指令包括最新发行版本的 RHEL/CENTOS 和最新的 Ubuntu

LTS 版本，支持其他操作系统的配置 Mesosphere 安装包仓库的文档可以在以下网站找到：https://mesosphere.com/downloads。

RHEL / CentOS 7

首先，你需要下载安装和配置系统使用 Mesosphere 库的 rpm 安装包，可以通过执行下面命令获取：

```
$ sudo rpm -Uvh http://repos.mesosphere.io/el/7/noarch/RPMS/
➥ mesosphere-el-repo-7-1.noarch.rpm
```

库安装完成之后，在要部署 Mesos masters 和 slaves 的主机上安装 Mesos 的安装包。在 masters 上，还需要安装 Mesosphere 的 ZooKeeper 安装包。执行如下命令：

```
$ sudo yum -y install mesos-0.22.2-0.2.62.centos701406
➥ mesosphere-zookeeper
```

在 slaves 上面，执行下面的命令：

```
$ sudo yum -y install mesos-0.22.2-0.2.62.centos701406
```

安装这些包的同时也会安装一些依赖包，如果这些你都已经安装完了，请跳到 3.3 节。

UBUNTU 14.04 (TRUSTY)

设置 Ubuntu 的安装库的话，需要先拿到 Mesosphere 的 GPG 公钥，它是用来做包的签名用的。然后需要将 Mesosphere 的库地址放到 Apt 的安装源列表，在系统上更新所有包的元数据。

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
$ echo "deb http://repos.mesosphere.io/ubuntu trusty main" |
➥ sudo tee /etc/apt/sources.list.d/mesosphere.list
$ sudo apt-get update
```

在拿到 Mesosphere 的公钥和准备好库之后，你可以在主备机上安装 Mesos 的包了，在主机上面你还得安装 ZooKeeper。执行下面的命令：

```
$ sudo apt-get install mesos=0.22.2-0.2.62.ubuntu1404 zookeeperd
```

在备机上面，执行下面的命令：

```
$ sudo apt-get install mesos=0.22.2-0.2.62.ubuntu1404
```

在安装这些包的同时 Mosos 和 ZooKeeper 也需要安装一些依赖包，如果这些你都已经安装完了，可跳到 3.3 节阅读。

3.2.2 从源文件编译并安装

尽管从 Mesosphere 提供的安装包中安装运行 Mesos 是目前最简单而快速的办法，我想最好是能从源代码编译一遍 Mesos。你可以从几个方面考虑这个：

- 你需要修改 Mesos 的安装配置或者激发其他额外的功能；
- 你喜欢建立自己的安装包，可能需要修改特定的地方；
- 你喜欢从 Apache 软件基金会获取源代码。

你可以通过各种不同的方法来配置、编译和安装 Mesos。下面的例子中，你可以看到使用 Mesos 配置脚本里面的默认配置。值得注意的是，默认配置中，编译 Mesos 也编译了一个捆绑版本的 ZooKeeper。

如果你想分开编译 ZooKeeper 或者使用其他的版本，可以优先配置 Mesos 到编译器上。可以在 Mesos 项目文档里找到一份完整的你可以传递给配置脚本的参数，地址是：http://mesos.apache.org/documentation/latest/configuration/。

注意：尽管我已经尽最大努力抓住重点并帮你过了一遍组建过程，但是指令会根据最新的发行版本而改变。可以在如下网站找到最新的指令：http://mesos.apache.org/gettingstarted/。

简单地说，编译 Mesos 需要如下条件：

- 一个 64 位的 Linux 发行版本；
- GNU 编译器 (GCC)4.4 以上版本或者 Clang3.3 以上版本。

提示：从 Mesos 0.23 版本开始，最小的编译器版本已经被更新到 GCC 4.8 和 Clang 3.5，并且支持所有平台。为了获得更多的信息，可以访问：https://issues.apache.org/jira/browse/MESOS-2604。

因为 Linux 的发行版本数量一直在增长，对我来说（或者 Mesos 作者）是不可能提供所有这些发行版本的安装和编译指令的。和前面小节的包安装一样，下面两个小节的编译安装指令也是提供 RHEL/CentOS 7 和 Ubuntu 14.04 LTS(Trusty)。

RHEL/CentOS 7 先决条件

幸运的是，RHEL 7 把 GCC 4.8.3 作为基础包的一部分供应了。这使得你在不需要更新编译器的情况下可以编译 Mesos 0.22.2 版本或者 0.23 以上版本。

所有安装 Mesos 的依赖包都可以在 RHEL 的 yum 管理工具里面找到，除了 Boto 这个包，它是提供接口调用 AWS 的一个 Python 模块。你需要用 Python 里面的一个叫 `easy_install` 的小工具来安装这个可选包，这个工具包含在 RHEL 里面。

要安装这些依赖包，需要在 Ubuntu 14.04 上编译 Mesos，执行下面的命令：

```
$ sudo yum -y groupinstall "Development Tools"

$ sudo yum -y install subversion-devel java-1.8.0-openjdk-devel zlib-devel
➥ libcurl-devel openssl-devel cyrus-sasl-devel cyrus-sasl-md5 apr-devel
➥ apr-util-devel maven python-devel

$ sudo easy_install boto
```

在这些依赖包被安装后，你可以跳到即将讲述的“编译”小节。

Ubuntu 14.04 LTS(Trusty) 的先决条件

也是比较幸运的是，Ubuntu 14.04 LTS 把 GCC 4.8.3 作为基础包的一部分供应了。这也使得你在不需更新编译器的情况下可以编译 Mesos 0.22.2 版本或者 0.23 以上版本。

所有的依赖包都在 Ubuntu 的 Apt 管理工具里面，包括 Boto 这个调用 AWS 的 Python 模块。

要安装这些依赖包，需要在 Ubuntu 14.04 上编译 Mesos，执行下面的命令：

```
$ sudo apt-get update

$ sudo apt-get -y install build-essential openjdk-7-jdk python-dev
➥ python-boto libcurl4-nss-dev libsasl2-dev maven libapr1-dev libsvn-dev
```

安装完这些依赖包之后，让我们来到下一个小节，编译 Mesos 和 ZooKeeper。

编译

在你安装好所有的依赖和开发工具之后，你可以下载一个 Mesos 的发行版本来开始它的编译过程。正如之前所说的，在编译过程中有很多选择。在例子里面，大多数都是选择默认配置，但是需要指定 `--prefix="/usr/local/mesos"` 这个选项，它确保了所有和 `Mesos` 相关的文件都存放在一个目录里面。你可根据自己的环境和喜好来修改这些选项。

执行下面的命令来下载、配置、编译 Mesos 0.22.2:

```
$ curl -L -O https://www.apache.org/dist/mesos/0.22.2/mesos-0.22.2.tar.gz
$ tar zxf mesos-0.22.2.tar.gz
$ cd mesos-0.22.2
$ mkdir build && cd $_
$ ../configure --prefix="/usr/local/mesos"
$ make
```

提示：你可以通过 `-j` 选项来开启可以同时启动的编译任务，从而加快编译时间。例如，在一个 4 核的机器上面你可以执行 `make -j4`，需要注意的是每个任务需要 2GB 的内存。

根据主机的核数情况，编译 Mesos 需要花费几分钟的时间。在编译完成之后，你可以跑里面包含的测试案例。这个步骤是可选的，但是如果你想用到 Mesos 源代码里面所有例子的 framework 的话，这确实是一个不错的注意。

运行测试案例的话，执行如下命令：

```
$ make check
```

你要观察测试案例的输出，在最后你可以看到类似下面这样的情况。

```
[==========] 539 tests from 86 test cases ran. (260794 ms total)
[  PASSED  ] 539 tests.
```

现在你已经在系统上编译并确保测试案例通过了，可以继续安装 Mesos 的步骤了。

安装 Mesos

在 Mesos 被编译好了之后，你可以执行下面的命令进行安装：

```
$ sudo make install
```

通过在配置脚本里面指定 `--prefix="/usr/local/mesos"` 这个选项，Mesos 会被安装到 /usr/local/mesos 这个目录里面。

让我们查看一下安装目录下面的子目录，知道它们的目的和内容：

- bin 包含了 Mesos 集群交互的命令行工具，例如 mesos-local、mesos-execute 和 mesos-ps。
- etc/mesos/ 包括 Mesos 集群的配置文件。
- include/ 包含和 Mesos 交互的 C++ 头文件。
- lib/ 包含本地的 Mesos 库文件，例如 libmesos.so。
- libexec/mesos/ 包含 Mesos 二进制文件和脚本。
- sbin/ 包含启动和停止 Mesos masters 和 slaves 的几个脚本，包括 mesos-master、mesos-slave 和 mesos-daemon.sh。
- share/mesos/webui/master/static/ 包含 Mesos 网页的静态代码。

值得注意的是，如果你有足够的工具使得你想在任何的 Linux 发行版本都能编译 Mesoss 的话，那么不可能让 Mesos 项目的维护者们提供给你所有的 Linux 发行版本的封装脚本。正如前面所述的一样，在 /sbin 下面的一系列脚本可以在前后台启动 `Mesos masters` 和 slaves。你应该自己根据操作系统的服务管理工具封装一个简单的脚本，调用 Mesos 提供的脚本来启动、停止和重启 `mesos-master` 和 mesos-slave。

提示：由 Mesosphere 提供的 Mesos 安装包中的脚本 mesos-init-wrapper 是一个开源工具，可以在这里找到：https://github.com/mesosphere/mesos-deb-packaging/blob/master/mesos-init-wrapper。

安装 ZooKeeper

当你在 build/ 目录里面编译 Mesos 的时候，同时也编译了 Mesos 维护者们捆绑在 Mesos 源码树里面的一个 ZooKeeper 版本。这个已经编译好了的 ZooKeeper（在 build/3rdparty 目录下），是可以被安装的了。

你可能需要将 ZooKeeper 重新放置到一个比较永久的目录里面，那么我们就继续将它复制到 /usr/local 下面，放到 Mesos 的旁边：

```
$ sudo cp -rp 3rdparty/zookeeper-3.4.5 /usr/local/
$ sudo chown -R root:root /usr/local/zookeeper-3.4.5
```

位于 ZooKeeper 的 bin 目录下面的几个脚本，协助你启动和停止 ZooKeeper 集群。你需要特别注意 zkServer.sh 这个脚本，它帮助你启动、停止和重启 ZooKeeper 集群。

就像前面所说的，让 ZooKeeper 项目的维护者们提供 ZooKeeper 可以运行的所有 Linux 发行版本的脚本是不可能的。一些基于 RPM 或者 Deb 操作系统的初始化脚本是放在 src/packages/ 目录下面的。当然你可以根据你自己的操作系统的需求，基于 zkServer.sh 去写一些适合你的系统的小型封装脚本。

现在你已经下载、编译、安装好了 Mesos 和 ZooKeeper，就要准备学习它们的各式各样的配置选项，还有适合两者的配置方法了。

3.3 配置Mesos和ZooKeeper

现在你知道如何部署 Mesos 的组件了，也安装了 Mesos 和 ZooKeeper，那你就需要配置它们，启动这些服务，使用这个集群。

配置 Mesos 和 ZooKeeper 的方法可能会稍微有一点不同，这取决于你之前是通过编译安装的还是通过安装包的形式安装的。不过不用担心，在下面的小节当中你两种都能学到。

3.3.1 ZooKeeper 配置

首先，你需要配置 ZooKeeper，因为 Mesos 集群的协调和领导者的选举需要它的参与。图 3.2 展示了 ZooKeeper 在你所部署的集群中的位置：

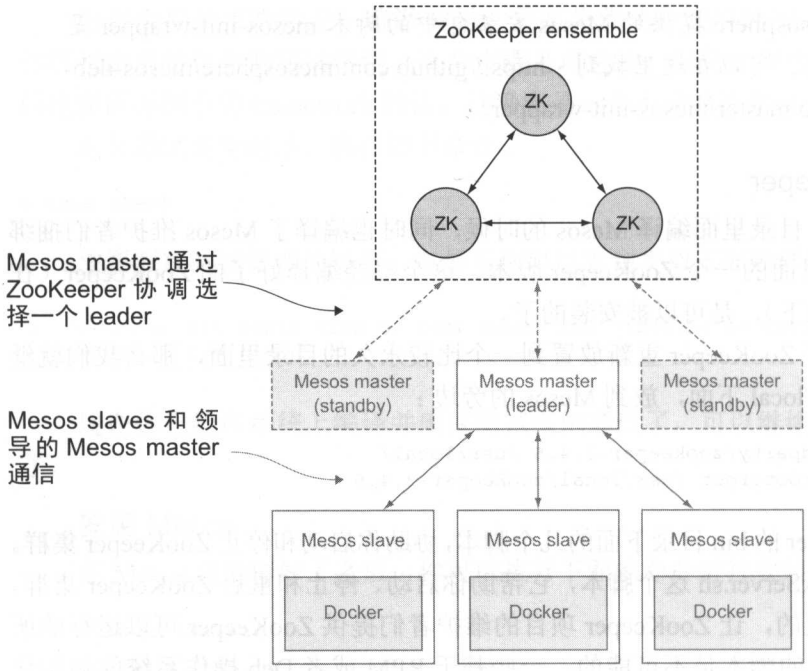


图 3.2 ZooKeeper 集群协调选举 Mesos master

配置文件 zoo.cfg 的位置取决于你之前是用编译安装的还是通过安装包安装的：

- 通过安装包安装的配置文件在 /etc/zookeeper/conf/zoo.cfg 下。
- 通过源代码编译安装的话，假设你将安装目录设置在 /usr/local/zookeeper-3.4.5，那么配置文件就在 /usr/local/zookeeper-3.4.5/conf/zoo.cfg 下。

让 ZooKeeper3.4.*x* 运行起来的话，需要的一些基本配置如下：

清单 3.1 基本的 ZooKeeper 配置

```
maxClientCnxns=50
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/lib/zookeeper
clientPort=2181

server.1=mesos-master-1.example.com:2888:3888
server.2=mesos-master-2.example.com:2888:3888
server.3=mesos-master-3.example.com:2888:3888
```

客户端最大的连接数

每一个 tick 的毫秒数

初始化同步阶段的 ticks 数目

发送请求和得到确认之间的 ticks 数目

快照存放的目录

客户端连过来的端口

指定这些服务器在同一个 Zookeeper 集群，第一个端口是追随者（follower）连接领导者 (leader) 的端口，第二个是用来选举的端口

除了创建 zoo.cfg 这个配置文件以外，你还需要为 ZooKeeper 集群中的每一个节点分配一个唯一的 ID。你会注意到这个清单中服务器被编号了：server.1、server.2，等等。在每一个 ZooKeeper 所在的主机上的 conf/ 目录下面创建一个叫做 myid 的文件，这个文件里面存放 1 到 255 的整数，代表着对应的每一个 ZooKeeper 节点。

提示：ZooKeeper 节点中的 myid 文件里面的数字是和 zoo.cfg 文件中的数字相对应的。

开启服务

尽管我只是启用了必需的最小的 ZooKeeper 配置需求，但是已经足够带起集群中的机器并让它们服务客户了。让我们继续在 masters 节点上启动服务：

- 对于包安装的执行 service zookeeper start。
- 对于源代码安装的执行 /usr/local/zookeeper-3.4.5/bin/zkServer.sh start。

当 ZooKeeper 服务启动后，你可以通过 Netcat 发送一个健康检查的命令来确保它处于健康状态。你可以发送一个 ruok 命令过去，它会回复一个 imok 的指示：

```
$ echo ruok | nc 127.0.0.1 2181
imok
```

如果上面的检查通过了，那就太妙了！这就说明 ZooKeeper 成功地运行并且可以为 Mesos 集群提供服务了。第 6 章包含了更多的 ZooKeeper 监控。但是现在既然它可以提供服务了，那么我们就继续前进，看一下 Mesos 的配置。

提示：如果需要更多关于 ZooKeeper 的选项配置，请咨询它的管理员手册，地址如下：http://zookeeper.apache.org/doc/current/zookeeperAdmin.pdf。

3.3.2 Mesos 配置

既然现在 ZooKeeper 集群已经搭建好了并且可以为 Mesos 提供服务了，我们就来看一下 Mesos 的配置。

这里提供的配置已经足够可以让集群来处理分布式负载了，但是不会包含 Mesos 提供的所有配置。你可以在如下网站找到 Mesos 的最新配置信息：http://mesos.apache.org/documentation/latest/configuration。

约定

配置部署 Mesos 需要几个约定：

- 基于文件的——当使用 Mesosphere 提供的安装包时，配置的值可以存放在磁盘上的文件当中，文件名是配置的选项。这样的例子包括 /etc/mesos/zk 和 /

etc/mesos-slave/attributes/rack。

- 基于环境的——对于包安装还是源代码安装，配置值可以被 mesos-master 和 mesos-slave 在启动时候从环境变量里面读到。这些值可以是环境变量的一部分，或者在 mesos 启动之前用 source 命令执行一个脚本得到。例如 MESOS_zk=” zk://...”。
- 基于命令行的——配置值可以在命令行中传到 mesos-master 和 mesos-salve 的二进制和服务脚本中，例如 mesos-master --zk=zk://...。

你会更多地把 Mesos 当作一个服务来访问，而较少把它当作一个命令行，所以本小节在介绍 mesos-master 和 mesos-slave 的时候使用的是基于文件和基于环境的方法。但是值得注意的是如果你的意愿很强烈的话，所有的这些配置选项都可以通过命令行来指定。

为了确定你安装的 Mesos 的配置文件路径，可以查看一下表 3.2。这个表假设你是通过编译安装 Mesos 的，路径是 /usr/local/mesos。

表 3.2 配置文件路径

操作系统	安装方法	配置方法	配置路径
RHEL andCentOS 7	Mesosphere 安装包	文件	/etc/mesos/ /etc/mesos-master/ /etc/mesos-slave/
RHEL andCentOS 7	Mesosphere 安装包	环境	/etc/default/mesos /etc/default/mesos-master /etc/default/mesos-slave
Ubuntu 14.04	Mesosphere 安装包	文件	/etc/mesos/ /etc/mesos-master/ /etc/mesos-slave/
Ubuntu 14.04	Mesosphere 安装包	环境	/etc/default/mesos /etc/default/mesos-master /etc/default/mesos-slave
All	源代码	环境	/usr/local/mesos/etc/mesos/mesos-master-env.sh /usr/local/mesos/etc/mesos/mesos-slave-env.sh

请根据你选的配置方法在上面的表格中选择路径。下面的这一小节包含了 mesos-master 和 mesos-salve 不需要引用配置文件的一些通常的选择。

Master 配置

为了让 mesos-master 能正常工作，有几个配置项是必需的，它们是：ZooKeeper 的 URL，Mesos master 的法定数目，Mesos master 的工作目录。图 3.3 告诉你现在处于部署整个集群的哪个阶段：配置 Mesos master。

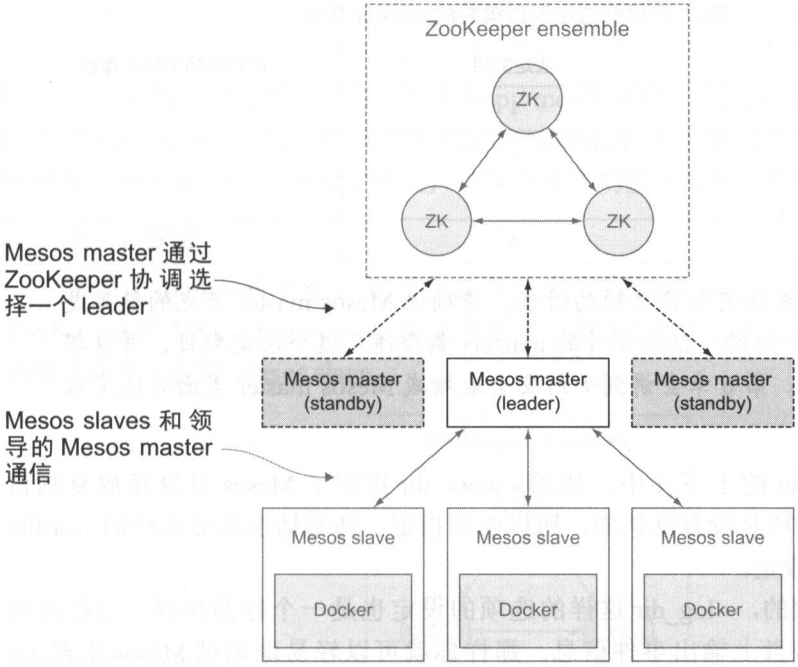


图 3.3 Mesos master 使用之前配置的 ZooKeeper 集群选举

选项——zk 是 ZooKeeper 的 URL，用来在 master 之间做领导者选举和协调。这个选项是在高可用的部署使用的，如果单独运行 Mesos 的话是不需要的。ZooKeeper 的 URL 使用下面的格式：

```
zk://mesos-master-1.example.com:2181,
➥ mesos-master-2.example.com:2181,
➥ mesos-master-3.example.com:2181/mesos
```

如果你在 ZooKeeper 集群里面启动了认证功能的话，可以在 URL 里面指定用户和密码，也可以在磁盘的文件上指定包含你认证细节的 ZooKeeper URL，如下：

```
zk://username:password@host1:2181,host2:2181,host3:2181/mesos
file:///path/to/zk_url
```

如果你是按照本章的指令来搭建 ZooKeeper 集群的话，现在不需要担心安全认证。第 6 章会讲到 Mesos 和 ZooKeeper 的安全性。

选项——quorum 指定了 Mesos master 集群里面的选举票数，并且用于 Mesos 复制注册。和 ZooKeeper 的一样，它应该按照表 3.3 那样去设置，*N* 是它的值。

表 3.3 根据法定数目需要的 master 数目

master 数目	法定数目	可容忍的失败机器数
1	1	0
3	2	1
5	3	2
2*N*-1	*N*	*N*-1

警告：当你配置法定数目选项的时候，请确保 Mesos master 节点的数目是和表 3.3 保持一致的。任何额外的 masters 都会违反这个法定数目，并且损坏复制的日志。第 6 章会讲到如何安全地增减 Mesos master 里面的法定数目。

在 Mesos master 的上下文中，选项 --work_dir 指定了 Mesos 自身存放复制日志的目录。这个选项是没有默认的，所以必须指定。通常情况类似这样的 /var/lib/mesos 是一个好的设定。

尽管不是必须的，--log_dir 这样的选项的设定也是一个好的抉择。这样就确保了 Mesos 会向磁盘上输出事件信息，那样你就可以轻易地调试 Mesos 集群了。Mesosphere 提供的安装包已经将 log_dir 设置在 /var/log/mesos 目录下面了。

提示：强烈要求设置另外两个选项：--hostname 和—ip。设置这两个选项可以确保你的 Mesos 服务使用到的主机和 ip 是正确配置，而不是自动发现的。这样的设置在多网卡主机上显得尤为重要。

基于前面所讲的例子，根据源代码安装 Mesos 的一个 mesos-master-env.sh 脚本如下：

```
export MESOS_zk=zk://mesos-master-1.example.com:2181,
➥ mesos-master-2.example.com:2181,mesos-master-3.example.com:2181/mesos
export MESOS_quorum=2
export MESOS_work_dir=/var/lib/mesos
export MESOS_log_dir=/var/log/mesos
```

至此在所有的基本配置完成后，你需要做的就是通过运行以下命令启动 Mesos master 脚本了：

```
$ sudo service mesos-master start
```

在 masters 的主机上将 mesos-slave 的服务关掉也是个好主意：

- On RHEL/CentOS 7------`sudo systemctl disable mesos-slave.service`;

- On Ubuntu 14.04 ----- `echo "manual" | sudo tee /etc/init/mesos-slave.override`。

你可以打开浏览器通过这个网址 http://mesos-master-1.example.com:5050 连接到一个 Mesos master 上面，用你刚才创建的 hostname 或者 ip 填补上面的域名。如果你连接到的不是当前主 master 的话，那么它会自动重定向到当时主 master 的。

Slave 配置

为了让 mesos-slave 能正常工作，有几个配置项也是你必须配置的，特别是 ZooKeeper 的 URL，Mesos slave 的工作目录。图 3.4 告诉你现在处于部署整个集群的哪个阶段：配置 Mesos slave。

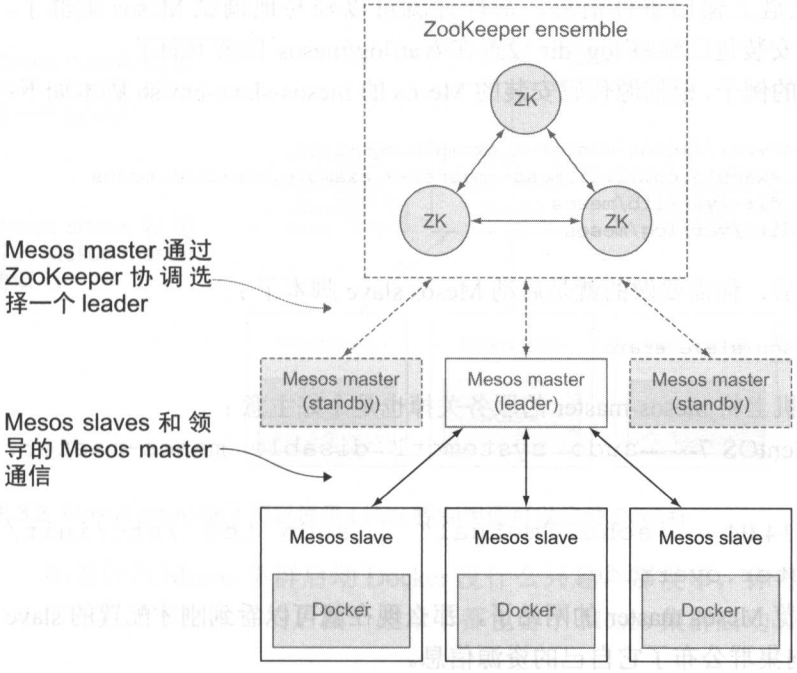


图 3.4 Mesos slave 用 ZooKeeper 检测并注册到主 master 里面

选项—master 指定了 ZooKeeper 的 URL，Mesos slave 通过它检测 leading master 和连接到整个集群当中。ZooKeeper 的 URL 使用下面的格式：

```
zk://mesos-master-1.example.com:2181,
➥ mesos-master-2.example.com:2181,
➥ mesos-master-3.example.com:2181/mesos
```

如果你在 ZooKeeper 集群里面启动了认证功能的话，你可以在 URL 里面指定用户和密码，也可以在磁盘的文件上指定包含你认证细节的 ZooKeeper URL，如下：

```
zk://username:password@host1:2181,host2:2181,host3:2181/mesos
file:///path/to/zk_url
```

注意：如果你是用 Mesosphere 提供的安装包的话，你应该在 /etc/mesos/zk 这里为 Mesos slave 的后台服务设置 ZooKeeper URL，而不是 /etc/mesos-slave/master。

在 Mesos slave 的环境中，选项 --work_dir 指定了 Mesos framework 的工作目录和沙盒所在。它是没有默认设置的，所以是必须设置的一个选项。通常会设置为 /var/lib/mesos，同时确保它不是放在那种类似 /tmp 的分区上面就更好了。

尽管不是必须的，--log_dir 这样的选项设定也是一个好的抉择。这样就确保了 Mesos 会向磁盘上输出事件信息，那样你就可以轻易地调试 Mesos 集群了。Mesosphere 提供的安装包已经将 log_dir 设置在 /var/log/mesos 目录下面了。

基于前面所讲的例子，根据源代码安装的 Mesos 的 mesos-slave-env.sh 脚本如下：

```
export MESOS_master=zk://mesos-master-1.example.com:2181,
➥ mesos-master-2.example.com:2181,mesos-master-3.example.com:2181/mesos
export MESOS_work_dir=/var/lib/mesos
export MESOS_log_dir=/var/log/mesos
```

基本配置完成后，你需要做的就是启动 Mesos slave 脚本了：

```
$ sudo service mesos-slave start
```

在 slaves 的主机上将 mesos-master 的服务关掉也是个好主意：

- On RHEL/CentOS 7------`sudo systemctl disable mesos-master.service`；
- On Ubuntu 14.04 ----- `echo "manual" | sudo tee /etc/init/mesos-master.override`。

如果你已经浏览 Mesos master 的网站了，那么现在就可以看到刚才配置的 slave 信息，并且 slave 向集群公布了它自己的资源信息。

像之前所说的那样，这里讲述的是能带起 Mesos 集群服务的最小配置。为了能得到 masters 和 slaves 的更多配置选项信息，请通过如下网址访问 Mesos 官方文档 http://mesos.apache.org/documentation/latest/configuration。第 4 章将讨论更多的 slave 的配置，包括属性和资源。

3.4 安装并配置Docker

由于多个应用程序的工作负载可以同时在任何的 Mesos slave 上面运行，每一个

执行器都在容器里面运行。除了 Mesos 自身的本地容器化外（利用 Linux 控制组），你也可以选择通过 Docker 来启动容器。这一小节介绍怎样部署 Docker，包括它的一些常用的配置选项。

图 3.5 告诉你现在处于部署整个集群的哪个阶段：在 Mesos slave 上面安装配置 Docker。

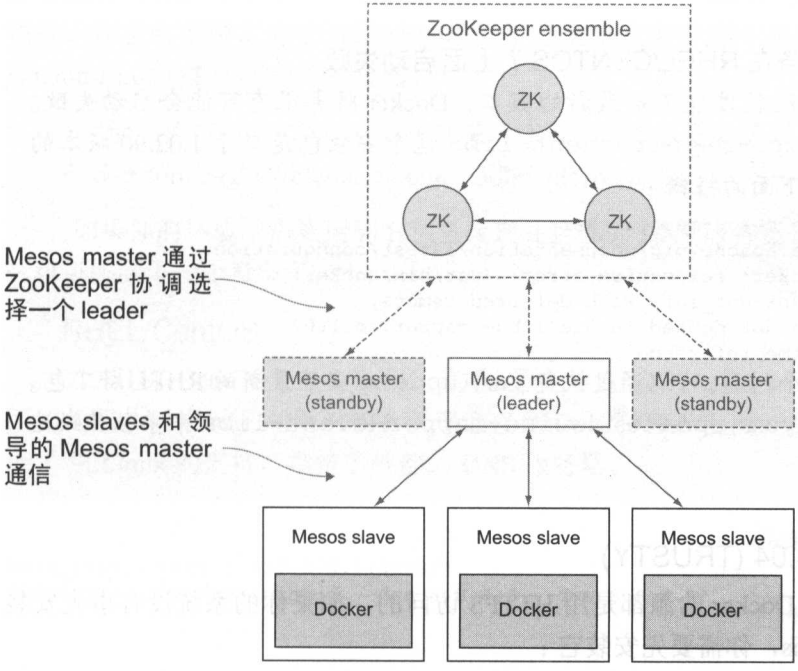


图 3.5 Mesos slave 除了可以使用 Linux 控制组还可以启动 Docker

如果你对 Mesos 集群启动 Docker 没什么兴趣，那没事，你可以后面再回到这一小节查看。不过本书的后面一些章节都包含了 Mesos 集群启动 Docker 的例子。

3.4.1 安装 Docker

就像我在第 1 章说到的，自定义执行器必须要在 Mesos slave 的主机上安装好。尽管 Docker 是一个 Mesos 的容器化，它也不是一个例外，也是需要在集群中能运行 Docker 容器之前将 Docker 安装配置好，并且运行起来了的。

RHEL/CENTOS 7

红帽在“extras”源上提供了安装包，所以你在安装 Docker 时并不需要安装其他的源。你可以用 yum 工具来安装它：

```
$ sudo yum -y install docker
```

对于 Docker 1.5.0，它的服务就不会在 RHEL 7 上自动运行，所以需要手工执行下面命令启动服务：

```
$ sudo service docker start
```

如果 Docker 服务在 RHEL/CENTOS 7 上面启动失败

如果你的系统软件包不是最新的版本，Docker 服务很有可能会启动失败。尤其是如果 device-mapper-event-libs 这个安装包是老于 1.02.90 版本的话，可能会看到下面的报错：

```
sudo service mesos-master start
http://mesos.apache.org/documentation/latest/configuration.
/usr/bin/docker: relocation error: /usr/bin/docker:
symbol dm_task_get_info_with_deferred_remove,
version Base not defined in file libdevmapper.so.1.02
with link time reference
```

为了解决这个问题，可以通过执行 yum update 安装最新的 RHEL 补丁包。也可以通过执行 yum update device-mapper-event-libs 来单独升级这个软件包。

UBUNTU 14.04 (TRUSTY)

Apt 上能下载 Docker 的源都是用 HTTPS 访问的，如果你的系统没有事先安装好 apt-transport-https，你需要先安装它：

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https
```

然后提取 Docker 安装包签名钥匙，将 Docker 安装源放到 Apt 的列表里面：

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv A88D21E9

$ echo "deb https://get.docker.com/ubuntu docker main" |
➥ sudo tee /etc/apt/sources.list.d/docker.list

$ sudo apt-get update
```

最后，安装 Docker 引擎包：

```
$ sudo apt-get -y install docker-engine
```

对于 Docker 1.6 之后，它的服务就会自动起来，可以执行下面命令检查 Docker 服务的状态：

```
$ sudo service docker status
```

3.4.2 配置 Docker

如果你需要在 Docker 后台服务里面做一些像设置一个代理访问，或指定 DNS 服务器之类的配置的话，现在就是最佳时间了。由于 Docker 使用默认的配置运行得很好，这里就不深入讨论 Docker 的配置了。但是我会讲述一些 RHEL/CentOS 7 和 Ubuntu 14.04 (TRUSTY) 常用配置。

提示：你可以在下面网站找到关于 Docker 的完整配置列表 https://docs.docker.com/engine/reference/commandline/daemon/。

如果你的环境不需要用到 HTTP 代理并且你的 DNS 服务器正常提供服务的话，你可以跳到下一个章节了。

RHEL/CentOS 7

在 RHEL/CentOS 7 上，Docker 的配置文件存放在 /etc/sysconfig/docker 中。下面的代码片段展示了利用 HTTP 代理连接到 Docker Hub 上面，在 Docker 主机上使能了 SELinux 的支持，指定了外部的 DNS 服务器。

```
http_proxy="http://127.0.0.1:3128/"
OPTIONS="--selinux-enabled --dns 8.8.8.8 --dns 8.8.4.4"
```

指定 HTTP 代理的地址

传递一些额外的选项给 Docker 后台服务

如果你改变了配置文件里面任何的配置，需要重启一下 Docker 服务：

```
$ sudo service docker restart
```

Ubuntu 14.04 (TRUSTY)

在 Ubuntu 14.04 (TRUSTY) 上，Docker 的配置文件存放在 /etc/default/docker 中。下面的代码片段展示了利用 HTTP 代理连接到 Docker Hub 上，在 Docker 主机上使能了 SELinux 的支持，指定了外部的 DNS 服务器。

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
export http_proxy="http://127.0.0.1:3128/"
```

利用 DOCKER_OPTS 传递参数给 Docker 后台服务

指定 HTTP 代理的地址

如果你改变了配置文件里面任何的配置，需要重启一下 Docker 服务：

```
$ sudo service docker restart
```

现在你既然可以正常启动 Docker 服务了，也将所需环境的特定配置处理好了，我们继续看一下对配置做哪些小调整可以让 Mesos 启动 Docker 容器。

3.4.3 配置 Docker 专用的 Mesos slaves

随着 Docker 的正常运行，你需要修改 Mesos slave 的可用容器列表，还有增加执行器注册的超时时间。通过增加执行器的注册超时时间，Docker 可以有更多的时间从 Docker Hub（或者你的私有仓库）里面获取镜像，防止 Mesos 认为某些事件发生而启动容器失败。这里会设置为 5 分钟。假设你是用 Mesosohere 安装包里面的基于文件的配置，那么你的配置命令可能会是这样的：

```
$ echo "docker,mesos" | sudo tee /etc/mesos-slave/containerizers
$ echo "5mins" | sudo tee /etc/mesos-slave/executor_registration_timeout
```

现在，重启 mesos-slave 服务让修改生效：

```
$ sudo service mesos-slave restart
```

注意：不像 Mesos 的默认容器，Docker 容器在 Mesos 里面是不占用磁盘配额的（对于 Mesos 0.22.2）。可以在 https://issues.apache.org/jira/browse/MESOS-2502 找到更多信息。

3.5 升级Mesos

相对于其他建立静态分区集群或者仅用一个 master 的分布式系统，Mesos 的容错架构使得升级 masters 和 slaves 的时候不需要任何的停机时间。Mesos 的接口甚至提供了一个方法在集群出现失败或者领导选举的时候通知到 framework，使得它们在有需要的情况下可以做出正确的反应。这一个小节解析升级 Mesos masters 和 slaves 的过程。

注意：在 Mesos 1.0 版本之前，它的维护者们只提供 *N*+1 的版本更新。也就是说，如果你的版本是 0.22，并希望更新到 0.24 的话，需要先更新到 0.23。更新升级时，最好是先去 Mesos 项目的页面上了解一下升级注意事项和升级文档。文档的地址在 http://mesos.apache.org/documentation/latest/upgrades。

对于任何一款软件，最终我们总会更新它的。无论是升级新的特性还是安装补丁防范安全攻击，软件升级同样会带来停机时间。不同的是，Mesos 的升级不会有集群离线的情况发生。

这一小节讨论 Mesos masters 和 slaves 的升级过程，升级其实也有和之前所讲的 Mesos 安装过程一样多的内容，还多了一些额外的考虑。第 4 章会解析为什么升级不需要停机时间这是可行的。

3.5.1 升级 Mesos masters

由于 Mesos masters 的数目会比 Mesos slaves 的数目少得多，最好有一些配置或自动化的管理工具来确保整个集群中法定数目的 masters 不会在同一时间升级。用一些类似于 Puppet 或者 Chef 这样的配置管理工具，可以让单个 master 在预定的时间表内升级，用 Ansible 或者 Fabric 这种自动化工具可以在多个 masters 之间串行升级。

升级 Mesos masters 需要执行下面的步骤：

1. 升级 Mesos master 二进制文件并且重启后台服务；
2. 升级调度器使用新的 Mesos 本地文件，JAR 包或者 egg；
3. 重启调度器。

3.5.2 升级 Mesos slaves

用著名的 *slave recovery* 和 *checkpointing* 的特性，Mesos 可以让 slaves 在升级的时候不中断运行任务。当 mesos-slave 的服务停止了，Mesos 的二进制文件在升级的时候，执行器——它们的任务可以继续运行。如果任务在 Mesos slave 恢复之前就已经正常完成了的话，这些任务会等待 `recovery_timeout` 这么久的时间，这个参数是 Mesos slave 的选项，默认是 15 分钟。但如果超过了这个时间，任何等待连到 Mesos slave 的任务都会自动终止。

提示：第 4 章会讲到 slave recovery 和 checkpointing 的更多细节。

当你在 Mesosframework 里面进行比较起伏的升级的时候，你最好准备一些配置管理或者自动化的工具。这将会确保集群中在任何时间内只有部分实例在升级，以免发生中途会终止升级的情况，或者升级伴随着其他任务的维护（例如打安全补丁）。升级 Mesos slaves 的话，你需要执行下面的步骤：

1. 升级 Mesos slave 二进制文件并且重启 slave 后台服务；
2. 升级调度器使用新的 Mesos 本地文件，JAR 包或者 egg（如果需要）。

3.6 小结

通过本章的例子，你应该已有一个合适功能性、高可用的 Mesos 和 ZooKeeper 集群准备好执行任务并启动 Docker 容器。需要谨记的事项如下：

- Mesos masters 使用 Apache ZooKeeper 来进行领导选举和协调沟通。Mesos slaves 和调度器利用 ZooKeeper 来检测 Mesos master 的领导者。
- 对于高可用的部署架构，你应该部署三个 ZooKeeper 实例，同时也需要三个 Mesos master 实例。对于简单的部署，你可以将 ZooKeeper 和 Mesos masters 一起分别部署在同一个机器上，当然如果你有其他的软件需要用到 ZooKeeper 做协调的话，可以将它和 Mesos master 分开部署。
- 你应该先安装配置 ZooKeeper，接着 Mesos master，最后到 Mesos slaves。
- Docker 是一个在数据中心运行分布式应用的好选择。尽管 Mesos 有对 Docker 的本地支持，但是 Docker 仍然需要单独安装。Mesos 需要微小的配置变动来支持 Docker。
- Mesos master 和 Mesos slave 都可以在集群不停机的情况下升级。
- Puppet 这个开源的配置管理工具，可以自动化配置和安装 Mesos、ZooKeeper 和 Docker 架构。它包含模块来管理 Mesos 部署的每一个组件。

第 4 章介绍更多的 Mesos 基本原理，包括 Mesos 怎样进行资源隔离、slave 的资源调用和容错。

4 Mesos原理

本章内容

- 资源分配，调度和保留
- 定制 Mesos slave 的资源属性和角色
- 利用容器隔离和监控资源
- 容错与高可用

现在你既然已经知道了 Mesos 的部署架构如何在一个单一通用的集群上面运营多个应用程序，那么我们继续深入探讨它是如何工作的。这一章会讲述 Mesos master 是如何处理资源的调度和分配的，一个工作负载的资源是如何被隔离和监控的，还有 Mesos 是如何提供容错技术和高可用环境来运行分布式应用的。

4.1 调度和分配数据中心资源

到现在为止，你已经学习了 Mesos 以资源提供的形式来为 framework 调度器提供可用集群资源。默认情况下这些资源包括可用的 CPUs、内存、存储和网络端口。在这一小节当中，你可以学到 Mesos 是怎样调度这些资源的，还有它是怎么分配模

块来将这些资源分配给不同的 framework 的。这一小节还介绍了怎样微调这些资源以适应你的环境需要的。

4.1.1 理解资源调度

Mesos 实现了两层的调度系统：Mesos slaves 提供它们可以利用的资源给 masters，通过 masters 再提供给调度器，调度器可以接受提供的所有资源，也可以接受一半，或者可以完全拒绝接受。让我们看一下图 4.1，这个图说明 Mesos slave 作为单一资源提供方将资源提供给了多个 framework 调度器：

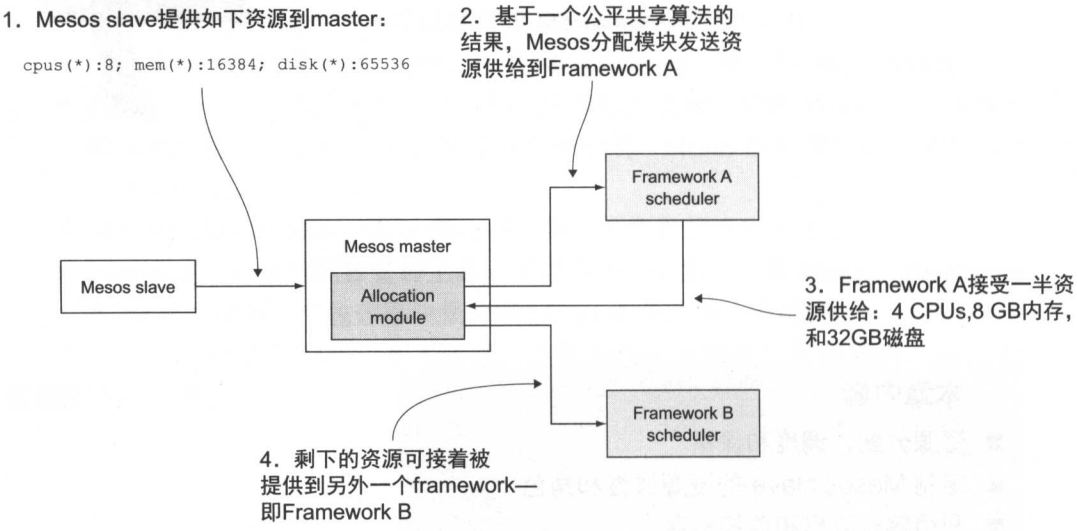


图 4.1 其他 framework 可以从资源提供方获取没分配的资源

在继续阐述之前，让我们看一下图 4.1 的情况：

1. 图中 Mesos slave 告诉它的 leading master 自身包含有 8 个 CPUs，16GB 的内存，64GB 的硬盘，星号 (*) 代表这些资源属于默认角色 (*default role*)，下一个小节会说到这个概念。
2. Mesos 的分配模块决定将整个资源信息通告给 framework A 的调度器。
3. Framework A 的调度器接受了一半提供给它的资源，预留下 4 个 CPUs、8 GB 的内存、32GB 的硬盘给其他应用程序。
4. Mesos 分配模块决定将剩下的所有未分配资源通知 framework B 的调度器。

这个过程会在 slaves 中当任务结束、资源可用时重复执行，基本上每几秒执行一次。

由于绝大多数的应用都是需要 CPUs、内存、存储和网络的，Mesos 事先定义好

了这些资源。让我们看一下 Mesos 提供给 framework 的默认资源有哪些？

默认资源

Mesos slaves 默认提供以下的资源：

- `cpus`——CPU 核数
- `mem`——内存
- `disk`——存储
- `ports`——网络端口

当 `mesos-slave` 启动的时候，空闲系统资源决定了上面各个资源的值，一小部分的资源用于 Mesos 本身的运行。

提示：内存和磁盘都是以 MB 来衡量的。

当 slaves 不停地向 masters 通告资源的时候，Mesos 的另外一部分——分配模块——负责决定哪个 framework 能得到资源提供。

4.1.2 理解资源分配

就像我上一小节提到的，Mesos 有一个分配模块决定了哪一个 framework 可以获取到资源。这些模块的可插拔性允许系统工程师实现自己的共享策略和算法以适应他们组织的需要。正如本书的第 1 部分所描述的，内置的配置模块利用优势资源公平（DRF）算法应该适合大多数 Mesos 用户的需求。

提示：想得到更多关于分配模块和分配算法的知识，可以访问 http://mesos.apache.org/documentation/latest/allocationmodule/。

Mesos 提供了几种方法在不需要替代或重新实施默认分配模块的情况下来微调资源调度。它们以角色（*roles*）、权重（*weights*）和资源保留（*resource reservation*）的形式出现。在图 4.2 中，你可以看到 Mesos slave 的资源可以保留成特定的角色，然后仅提供给以该角色注册的 framework。

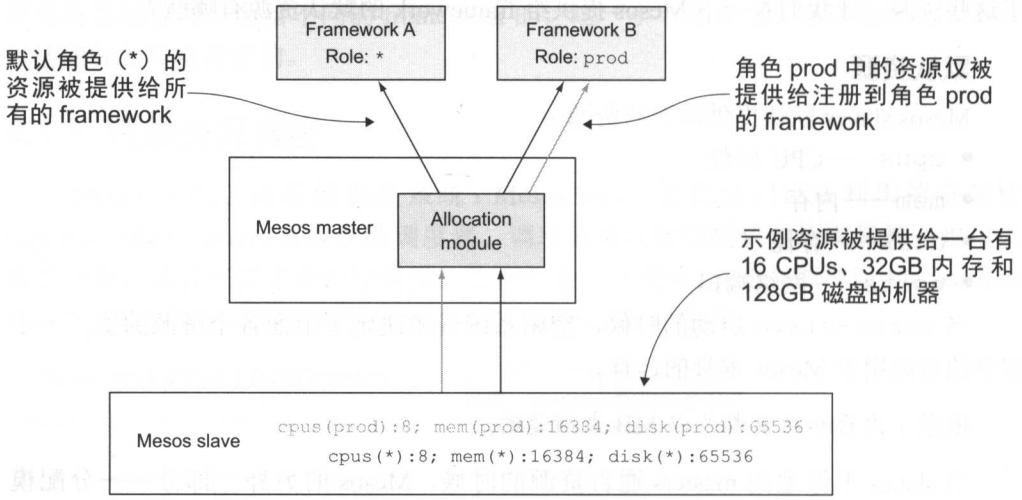


图 4.2 Mesos slave 资源可以为 framework 保留成特定角色

通过组合角色、权重和资源保留，你可以为特定的应用提供可用资源的保证，也可以控制为这些应用程序提供资源的频率。下面几个小节介绍更多的这些概念。

角色

Mesos 集群里面的角色概念可以让你以任意组的形式来组织 framework。为了在 Mesos 集群中用角色这个特性，你需要在 masters 中配置一串集群中准备要用的静态角色列表。通过设置—role 这个配置项一个值，下面的例子可以让 framework 在一个数据中心注册三种角色，开发、测试和生产。

```
--roles="dev,stage,prod"
```

然后 framework 可以在注册到 Mesos master 的时候指定其中一个角色。这样就可以使得多个团队或多个环境共享一个大型 Mesos 集群了，而不是分散地拆分成多个小集群。当需要决定提供给哪个 framework 哪些资源的时候，Master 就会动态地计算每个角色占用的资源。你也可以用角色来确保一些特定的工作负载跑在一些特定的子网上面，例如负载均衡器或者反面代理在边缘网络节点上面运行。

除了在 framework 上面指定角色，Mesos 也可以通过资源保留来指定资源属于哪个角色，这些你都可以在后面的小节学习到。

权重

除了角色之外，集群可以配置每个角色的权重，作为一种区分角色优先级的方法。当决定向哪个 framework 优先提供资源的时候，Mesos 会向低于权重公平份额最多的一个 framework 提供资源。

用上一节用到的同样的角色——dev、stage 和 prod，你可以这样配置 masters 使得 prod 角色的优先级高于 dev 和 stage：

```
--weights="dev=10,stage=20,prod=30"
```

要在实践中理解这个，可以用 prod 的权重作为例子。角色 prod 所在的 framework 得到的资源会是 dev 角色所在 framework 的 3 倍。当一个新的资源通知 masters 的时候，分配模块检查集群中的角色，决定其中谁的资源是提供的权重份额中最低的。然后检查角色中有哪些 framework，并提供资源给提供的权重份额中最低的。

保留资源

尽管权重的方法可以使得某些特定的角色比其他角色获取到更多的资源，Mesos 也提供了资源保留的方法。保留可以确保某些角色可以获取一定量的 slave 资源，但是要付出代价：这样做会削弱集群的利用率。

假设某一时刻你有一台 16 核，32GB 内存，128GB 硬盘的机器。你总是想让一半的资源（8 核，16GB 内存，64GB 硬盘）可以分配到注册了 prod 的 framework 中。这样的配置可以通过下面的方式在 Mesos slave 配置：

```
--resources="cpus(prod):8; mem(prod):16384; disk(prod):65536"
```

剩下的任何资源（8 核，16GB 内存，64GB 硬盘，假设没有系统资源开销），会分配给默认角色 (*) 的 framework 和那些注册进 master 的时候没有指定一个角色的 framework。

提示：Mesos 中默认的角色用星号 (*) 表示，这你可以用 Mesos slave 的选 --default_role 来做自定义配置。

现在我们已经讨论了怎样定制所有 Mesos slave 提供的默认和预先定义的资源，但是 Mesos 也允许你配置单个 slave 的资源，包括 CPUs 核数、内存数量，或者增加一个新的资源。

4.1.3 定制 Mesos slave 资源和属性

在前一小节，你学到了在 slave 上配置让资源分配给特定的角色和保留资源的方法。但是怎样创建自定义资源和重写默认值呢？像很多其他的东西一样，在 Mesos 里面的资源也是可以定制的。如果你需要新增一个资源（例如 ephemeral_ports，后面会介绍的），或者硬编码你的 CPU 和内存数量，或者允许其他一些服务在 slave 上有一点运行的空间的话，那么是非常有用的。

默认情况下，Mesos 在资源提供清单里面通告了处理器、内存、硬盘和端口。让我们看看怎样在 slave 中进行定制资源。

定制 slave 的资源

Mesos 提供了三种不同类型的资源：标量（scalars）、范围（ranges）和集合（Sets）。这里有三种类型的例子：

- 标量 --- 处理器资源的值是 8；内存资源的值是 16384。
- 范围 --- 端口值的范围是 10000 到 20000。
- 集合 --- 磁盘资源的值有 ssd1、ssd2 和 ssd3。

注意：一个没有处理器和内存资源的 Mesos slave 是不能给 framework 提供服务的！

由于在讲资源保留的时候讲过了定制 slave 的资源，这里就不太深入讲了。你可以用 --resources 这个配置选项来为 slave 中默认的或指定的角色自定义资源：

```
--resources="cpu(*):4; mem(*):8192; disk(*):32768; ports(*):[40000-50000];
➥ cpu(prod):8; mem(prod):16384; disk(prod):65536"
```

定制 slave 属性

除了可以定制 slave 中可消耗的资源外，还可以指定 slave 的属性。这些可选属性是任意键值对，可用于提供机器数据给 master 和 framework。本书在写的时候，所有的属性值都是以文本的方式出现的。

下面的例子通过 --attributes 的配置选项制定了机器的数据中心、支架、操作系统和可用的 Python 版本：

```
--attributes="datacenter:pdx1; rack:1-1; os:rhel7; pythons:python2,python3"
```

现在在 slave 向 master 提供的清单列表里面（或者 master 向 framework 提供的清单列表），你都可以看到这些属性了。这样可以让你在异构的环境中做出对调度工作更明智的决定。

提示：获取更多关于 Mesos slave 资源和属性的信息，请访问 http://mesos.apache.org/documentation/latest/attributes-resources。

4.2 使用容器隔离资源

容器是在基础架构当中提升效率的一种极佳方式。和虚拟机相比，容器更轻量级，并且它可以允许你的应用程序和代码和其他的工作负载相隔离。Mesos 的一个

基本理念就是，使用容器隔离进程是利用计算机资源的最有效方式。

非常好的是，Mesos 实现了在写本书时候的两个最有名的容器 Linux LXC 和 Docker。通过在容器内运行执行器和任务，Mesos slave 可以让多个不同的 framework 执行器相互运行而不受影响。这就好像虚拟化管理程序可以在一个物理主机上面运行多个虚拟机一样，不同的是容器的启动比起整个操作系统的启动是更轻量级的。

Mesos 有一个基本的组件叫做 containerizer。在写本书的时候，Mesos 包括两个 Containerizers，可以通过在 slave 上配置 --containerizers 配置选项，它们是 mesos 和 Docker。mesos containerizer 的任务是通过 cgroups 隔离工作负载，监控资源消耗的情况，Docker 则调用了 Docker 容器的运行时，并且让你在 Mesos 集群当中启动你之前制作好的镜像。

> **提示**：Mesos 也提供了外部的 containerizer 接口，让你实现一个新的容器规范。也许你不需要编写你自己的 containerizer，所以我这里就不详细讲解了。如果你真的非常感兴趣学习外部 containerizer 接口，想为 Mesos 创建一个新的容器规范的话，可以访问如下网站 http://mesos.apache.org/documentation/latest/external-containerizer/。

除了 Containerizers，Mesos 提供了多种集中资源隔离的方法。例如默认的 `posix/cpu` 和 `posix/mem`，提供了资源监控的方法。其他的，例如 `cgroups/cpu` 和 `cgroups/mem`，提供了通过 Linux 内核的 cgroup 进行的资源隔离和数量定额控制的方法。下面的几个小节会浏览一下 Mesos 资源隔离的几个方法。

4.2.1 隔离并监控 CPU、内存和磁盘

为了确保 Mesos slave 上面的一个工作负载不会影响另外一个工作负载，Mesos 提供了几种不同的资源隔离方法。就像之前提到的一样，Mesos 支持 LXC、Docker 和基本的 POSIX 可兼容的操作系统的资源监控（不是隔离）。

让我们看看怎样隔离和监控 CPU、内存和磁盘的资源使用。Containerizer 的隔离方法是在 slave 的配置选项 --isolation 当中配置一个逗号分隔的列表。

Linux 的资源隔离

在 Linux 上可以使用的资源隔离方法如下：

- `cgroups/cpu` 和 `cgroups/mem`，通过 Linux 内核的 *control groups* 来隔离 CPU 和内存资源。
- `filesystem/shared` 通过在容器里面和宿主主机本地磁盘配对一个目录，确保容器有一个可以读 / 写或者只读的私人工作目录。这可以通过 framework

来指定，或者和 --default_container_info 一起使用。将容器里面的 /tmp 目录匹配到沙盒工作目录的例子如下：

```
{
  "type": "MESOS",
  "volumes": [
    {
      "host_path": "private",
      "container_path": "/tmp",
      "mode": "RW"
    }
  ]
}
```

- namespaces/pid 使能进程号命名空间，确保容器看不到其他容器的进程。
- posix/disk 是一个可以监控容器使用的 Linux 存储隔离器。通过在 slave 上面设置 --enforce_container_disk_quota，可以确保单一容器使用的磁盘不会超过分配给它的最大份额。
- posix/cpu 和 posix/mem，这是 Mesos 默认的隔离器，仅提供 CPU 和内存资源的监控。更多关于这两个隔离器的信息可以关注下一个小节。

注意：当使用 filesystem/shared 隔离器的时候，容器里面的挂载点不能是文件系统的一部分。例如，当 Mesos slave 的 work_dir 是 /tmp/mesos 的时候，那么 container_path 不能是 /tmp。

当你查询 slave 的 /monitor/statistics.json HTTP API 地址的时候，你可以获得类似下面这些容器的数据：

- cpus_system_time_secs
- cpus_user_time_secs
- mem_anon_bytes
- mem_file_bytes
- mem_mapped_file_bytes
- mem_rss_bytes

通过 cURL 命令，你可以按照如下的方式获取到 leading Mesos master 的数据：

```
$ curl -s http://mesos-leader:5050/monitor/statistics.json |
➥ python -m json.tool
```

提示：第 6 章会包含监控 Mesos 集群的不同方法。

尽管在写本书的时候 Linux 为 Mesos 提供了最多的资源隔离的方法，但也可以在其他 POSIX 兼容的操作系统中监控资源使用，例如 Mac OS X。

在其他 POSIX 系统的资源监控

下面的资源隔离方法在其他的 POSIX 系统中是可用的，虽然这些系统 Mesos 可能不支持资源隔离，但肯定是支持资源监控的。像 Mesos 0.22.2 版本，POSIX 兼容的非 Linux 操作系统受限于如下的隔离器：

- `posix/cpu` 和 `posix/mem`，仅提供 CPU 和内存资源的监控，没有隔离性。

目前，这些隔离器提供了每个执行器（每个进程）的 CPU 和内存资源监控，类似于执行 ps 命令的输出一样。由于 Mesos 会减去 slave 的资源提供里面已经分配的资源，所以通常如果不是你的非 Linux 集群资源利用率达到 100% 的话都是没有问题的。但是如果一个任务超过了分配给它的内存（例如，内存泄漏或者错误配置了堆内存），那么在缺少资源隔离的情况下就会影响在同一个 slave 下面运行的其他任务了。有了 Linux cgroups 的资源监控的话，它就会调用内存溢出杀手（OOM killer）来停止违规的任务了。

4.2.2 网络监控和限速

如果你对单一容器上的网络监控和出口流量限制感兴趣的话，Mesos 提供了一个你也许觉得有用的可选网络隔离器。在写本书的时候，在 Mesos slave 上面通过 JSON 格式的接口提供统计数据。网络监控对容器来说是透明的，所有的容器都会持续共享 Mesos slave 的公共 IP。

网络监控和速率限制相对来说是一个比较新的特性，这在 Mesos 0.20 中有介绍，利用了现代 Linux 内核已有的功能。下面几个小节会向你展示怎样编译带有网络隔离器的 Mesos，还有怎样监控在 Mesos slave 上面运行的任务的网络流量。

编译带有网络隔离器支持的 Mesos

默认情况下，Mesos 没有安装网络隔离器，在写本书的时候也没有在 Mesosphere 提供的安装包里面使能。你可以在安装编译配置的时候通过 --with-network-isolator 这个参数来使能它。对于 Mesos 0.22.2 版本，需要在 Linux 内核 3.15 以上才能使用，并要有前提要素和额外的设置。

一个 Linux 内核版本和补丁的重要注意事项

为了可以让每个容器上的网络监控可以工作，在你运行 Mesos slave 的 Linux 内核上面需要实施几个关键的补丁。

在写本书的时候，RHEL/CentOS is 7.1 的发行版本使用的是内核 3.10，所需要的补丁红帽还没有发布。不幸的是，除了重新编译之外，没有更直观的升级内核的方法，但是重新编译已经不在本书的范围之内了，当然也是红帽不支持的。就 CentOS 而言，你可能会发现 elrepo 有用：http://elrepo.org/tiki/tiki-index.php。

在Ubuntu 14.04.3 LTS上面，内核版本已经是3.19了。如果你运行老的LTS版本（类似于14.04.2）的话，可以使用安装包管理器来升级内核。执行如下命令，然后重启来加载到新的内核：

```
sudo apt-get -y install linux-image-generic-lts-utopic
```

如果你不是用到上述这些操作系统或发行版本，一定要确认安装内核3.15以上的，它包含了一些必需的网络补丁。如果你要用到比3.15老的内核，需要手工去执行下面的补丁，并且重新编译内核。补丁可以在git.kernel.org上面找到：

- 6a662719c9868b3d6c7d26b3a085f0cd3cc15e64
- 0d5edc68739f1c1e0519acbea1d3f0c1882a15d7
- e374c618b1465f0292047a9f4c244bd71ab5f1f0
- 25f929fbff0d1bcebf2e92656d33025cd330cbf8

除了使用最新Linux内核外，你需要一个新版本的Netlink协议库套件，也被称为*libnl*。而要使用网络隔离器，则Mesos需要3.2.25以上的libnl版本。

编译Netlink协议库套件

在写本书的时候，在RHEL 7和Ubuntu 14.04.3LTS上面的Netlink协议库套件最新的是3.2.21。为了能让Mesos使用网络隔离器，你需要到www.infradead.org/~tgr/libnl这个地方去下载libnl3。安装libnl3需要Bison和Flex，可以在REHL7、CENTOS7和Ubuntu 14.04的系统安装包里面安装它们。在所有这些前提要素都准备好的情况下，执行下面的命令：

```
$ curl -LO http://www.infradead.org/~tgr/libnl/files/libnl-3.2.25.tar.gz
$ tar zxf libnl-3.2.25.tar.gz
$ cd libnl-3.2.25
$ ./configure
$ make
$ sudo make install
$ sudo ldconfig
```

默认情况下，libnl3配置了/usr/local/作为前缀，这就意味着它的头文件信息安装在/usr/local/include/libnl3/netlink下面。现在可以继续编译带网络隔离器的Mesos了。

在根据第3章的必备构建指示安装所需工具来编译Mesos后，执行下面的命令来编译带网络隔离器的Mesos。如果你已经构建了一次Mesos的话，在你重新编译之前一定要执行一遍make clean命令。

注意：由于在Mesos 0.22.2的配置脚本里面有一个bug，你需要覆盖

Makefile 中一个叫 LIBNL_CFLAGS 的选项。默认情况下，libnl3 将它的头文件安装在 /usr/local/include/libnl3 里。我已经将解决方法放到下面的命令中，如果想得到更多的信息，可以访问 https://issues.apache.org/jira/browse/MESOS-1856。

```
$ ../configure --prefix=/usr/local/mesos --with-network-isolator
$ make LIBNL_CFLAGS="-I/usr/local/include/libnl3"
```

在 Mesos 编译完成后，执行第 3 章说到的一些剩余的安装配置指令。

配置主机

当使能了每个容器的网络监控时，Mesos slave 上面的每个容器使用了 Linux 内核的网络命名空间，每一个容器有一个独立的网络堆栈。使得主机上面的容器可以进行网络监控的第一件事就是限制操作系统分配临时端口的范围，这样可以为容器保留临时端口。

注意：本小节提供的指令是一个例子。基于你部署的软件和所使用的端口，你的环境可能是一个不同的网络配置。你可能对于 Mesos 项目文档里面关于网络监控的部分感兴趣，地址为：http://mesos.apache.org/documentation/latest/network-monitoring。另外，如果你对于 Linux 内核的网络命名空间很感兴趣的话，下面的 LWN 文章是一个很好的信息来源：http://lwn.net/Articles/580893。

修改主机的临时端口范围，向 /etc/sysctl.conf 这个文件追加下面的命令，并且重启主机：

```
net.ipv4.ip_local_port_range = 57345 61000
```

尽管你也可以用 sysctl –p 来配置，重启机器能确保：任何使用了老范围分配临时端口的服务都会使用新的端口范围，避免了可能的端口冲突。如果上述命令执行成功，你可以看到类似下面的结果：

```
$ cat /proc/sys/net/ipv4/ip_local_port_range
57345 61000
```

配置 Mesos slave

然后你需要配置 Mesos 来使能网络监控和提供 ephemeral_ports 资源——保留给容器的端口范围。具体来说，包括下面的事项：

1. 追加 network/port_mapping 到资源隔离清单；
2. 修改 slave 的资源，提供一定范围的临时网络端口给容器使用；

3. 为每个容器的正常网络流量配置一个临时端口。

上面这些事项点的配置类似如下：

```
--isolation="cgroups/cpu,cgroups/mem,network/port_mapping"
--resources="ports:[31000-32000];ephemeral_ports:[32768-57344]"
--ephemeral_ports_per_container=1024
```

注意：确保 ephemeral_ports 的资源和提供给主机的操作系统的临时端口范围没有重叠！

如果你想限制每一容器的出口流量，可以在 Mesos slave 上面配置 --egress_rate_limit_per_container 这个选项，它是用字节来作为单位的。

提示：传递给 egress_rate_limit_per_container 的参数在 Mesos 的代码里面是以字节 *object* 为单位的。因此以下的值是有效的：10B，10KB，10MB，10GB，10TB。

下面的 slave 配置例子限制了每个容器的出口流量是 10Mbps：

```
--egress_rate_limit_per_container=12500KB
```

获取容器网络数据

在带有 network/port_mapping 隔离器的 mesos-slave 后台程序启动之后，你可以获取到运行中容器的如下网络字段信息（除了 Mesos 默认提供的数据信息外）：

- net_tx_bytes 和 net_rx_bytes
- net_tx_dropped 和 net_rx_dropped
- net_tx_errors 和 net_rx_errors
- net_tx_packets 和 net_rx_packets

这些数据作为 Mesos slave /monitor/statistics.json 接口的一部分，可以这样查找：

```
$ curl -s http://slave1:5051/monitor/statistics.json | python -m json.tool
```

现在你已经知道了 Mesos 是怎样调度和分配资源的，并且知道了它是怎样用容器来隔离单一主机上的工作负载的，让我们继续看看它是怎样在容错和高可用情况下进行这些工作的。

4.3 了解容错和高可用

在设计上，Mesos 为运行的应用提供了一个容错的环境。Mesos 服务——master 后台进程和 slave 后台进程——利用了分布式和高可用的方法，确保一个组件都不

能中断整个集群。让我们看一下图 4.3，它解释了不同 Mesos 组件之间的容错和高可用属性。

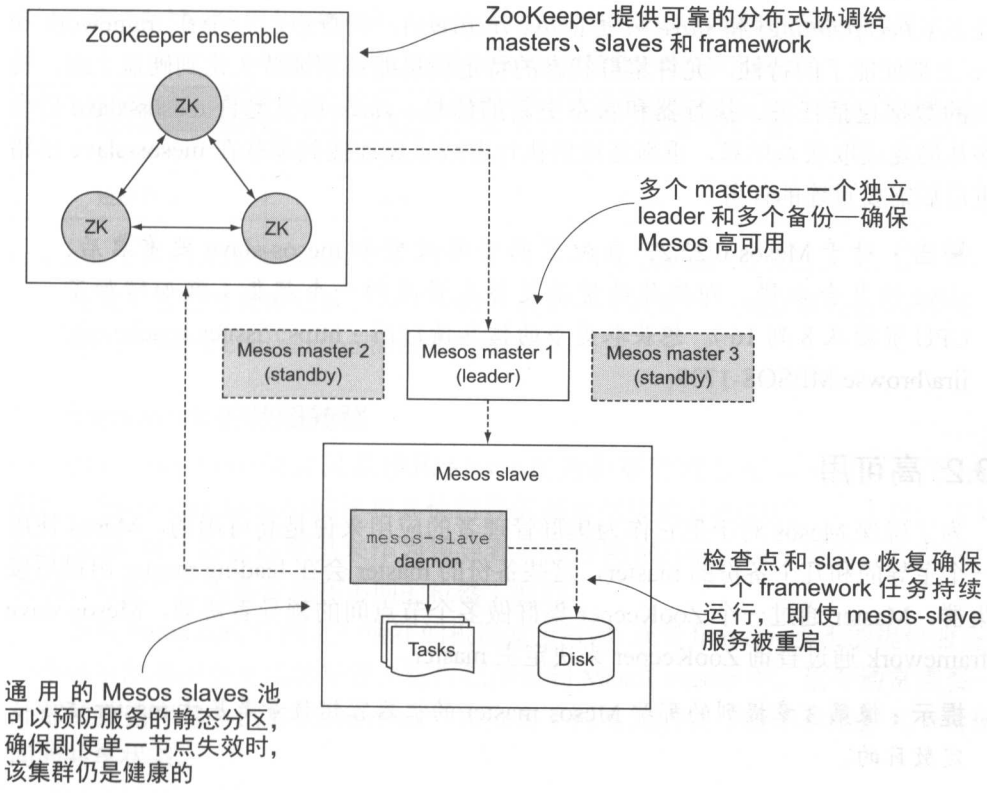


图 4.3 不同 Mesos 组件之间的容错和高可用

容错 VS 高可用

术语**容错**和**高可用**是密切相关的，并经常互换使用。但它们是两个不同的概念，在继续阐述之前，让我们来了解一下本小节是如何定义使用它们的：

容错——一个系统在它的一个或多个组件出错后，仍然能从容地处理并提供服务的能力。

高可用——一个系统能够长时间提供服务的能力，致力于 100% 提供服务的目标。

这一小节介绍不同的 Mesos 组件 ---framework、master 和 slave---- 在出错中恢复的情况。

4.3.1 容错

为了能从容地处理出错，Mesos 实现了两种特性（两种都是默认使能了的）：检查点 (*checkpointing*) 和 *slave* 恢复 (*slave recovery*)。检查点，一个在 framework 和 slave 上都使能了的特性，允许集群状态的特定信息能定期地持久化到硬盘上面。检查点的数据包括任务、执行器和状态更新的信息。slave 恢复允许 mesos-slave 后台程序从磁盘读取状态信息，重新连接到执行器或重新连接到那些在 mesos-slave 出错或重启后需要重连的任务。

> **警告**：对于 Mesos 0.22.2，在配置的资源改变和 mesos-slave 被重启后，slave 恢复会出错，即使你的资源是原先资源的一个超集（例如增加了 CPU 资源从 8 到 16）。想获取更多的信息请访问：https://issues.apache.org/jira/browse/MESOS-1739。

4.3.2 高可用

为了确保 Mesos 对于把它作为集群管理者的应用来说是高可用的，Mesos 使用了一个主导的和几个备份的 master，这些备份的 master 会在 leading master 出错后接管集群。Master 通过一个 ZooKeeper 集群做多个节点间的领导者协调，Mesos slave 和 framework 通过查询 ZooKeeper 来决定主 master。

> **提示**：像第 3 章提到的那些 Mesos master 的机器容错数量是基于 master 法定数目的。

通过检查点，slave 恢复，多 masters 和 ZooKeeper 的协调等这些机制，Mesos 集群能够在出错的情况下不影响整体集群的健康。正是由于这些对出错的从容处理，Mesos 才得以在没有停机的情况下升级。让我们现在就看看集群是怎样处理出错和升级的吧。

4.3.3 处理出错和升级

许多事件通常会导致基础设施的停机时间和中断，包括网络分区、机器故障、停电等。本小节的目标是使你能够知道 Mesos 在三种潜在的出错场景中的容错和高可用：

- 机器故障 ------ 底层物理机或者虚拟机宕机；
- 服务 (进程) 退出 ---------`mesos-master` 或 `mesos-slave` 后台进程退出；
- 升级 -------`mesos-master` 或 `mesos-slave` 升级和重启。

幸运的是，Mesos 和 Mesos framework 有能力处理上面的每一个出错模块。表 4.1 列出了上述不同出错场景的 Mesos 组件是否也会出错的情况：

表 4.1 Mesos 的 framework、master 和 slave 在典型的出错场景中是怎样处理的。（yes/no 代表指定的组件在出错场景中是否被支持）

出错场景	framework 故障转移	Master 故障转移	Slave 故障转移
机器故障	yes	yes	no
进程退出	yes	yes	yes
升级	yes	yes	yes

注意：在进行升级之前，确保查看了最新的升级文档。地址在 http://mesos.apache.org/documentation/latest/upgrades。

framework 的故障转移

因为 *framework* 仅仅只是使用 Mesos 作为集群管理分布式应用的一个 Mesos 组件，framework 的故障转移和其他组件的高可用架构是相似的：一个单一实例的 framework 调度器被选举出来作为领导者并注册到 Mesos master 里面，其他的几个实例，可能是作为备份跑在不同的机器上面的。

如果 framework 实例之一所在的机器发生故障，或者 framework 不能提供服务了，一个备份的实例会变成领导者并将自己注册到 Mesos master 中。这个通常需要一些外部的数据存储（例如 ZooKeeper）来协调不同实例的领导选举和维护几个实例之间的共享状态。

提示：可以在以下网址得到 ZooKeeper 领导选举的更多信息 http://zookeeper.apache.org/doc/current/recipes.html#sc_leader-Election。

让我们看看图 4.4，它展示了 framework (framework) 故障转移的一个类似过程。

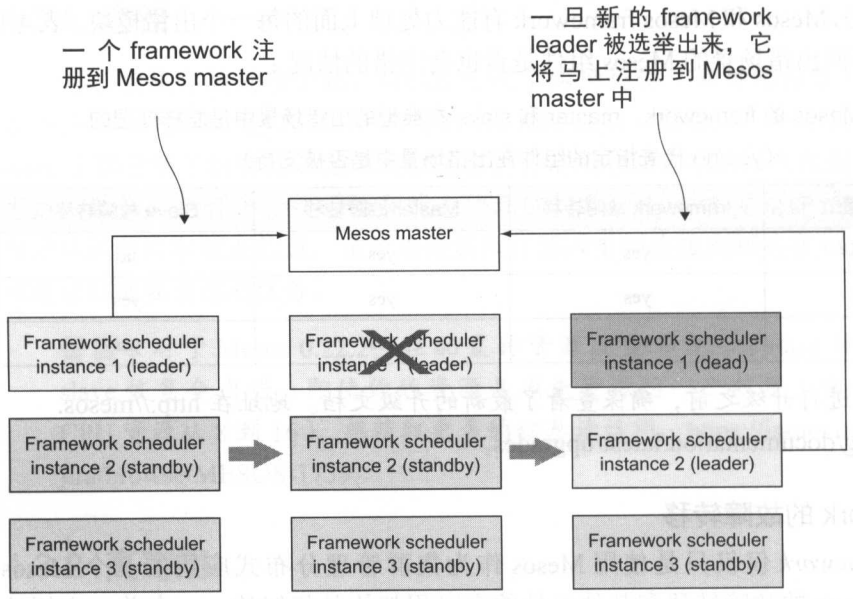


图 4.4 一个高可用的 framework 调度器故障后，另一个可以在不打扰运行任务的时候注册到 Mesos master 里面

让我们看看图中的各个事件：

1. 由之前的领导选举决定，framework 调度器实例 1 是当前的领导者，它注册到 Mesos maste 里面了。
2. framework 调度器实例 1 变成离线状态。
3. 在 framework 调度器实例中发生领导选举，新的领导者用同样的 ID 注册到 Mesos master 里面，恢复了正常的工作。

这里假设 framework 调度器是从开发到部署都是允许高可用设计的。仅仅由于 Mesos 允许 framework 注册到 Mesos master 里面，并不意味着 framework 在它的调度器故障之后也可以这样做。但是 Mesos 的确提供了调度器在主 Mesos master 发生故障后，注册进新的主 master 的方法。让我们在下一小节看看。

Master 的故障转移

Mesos masters 都是用 ZooKeeper 作为领导选举的，其中之一作为领导者，剩下的作为备份或待命的 masters。Mesos slaves 和 framework (framework) 同样是用 ZooKeeper 来决定哪个是主 master。如果 Mesos master 发生故障（无论是机器故障还是服务出错），framework 和 slaves 检测到它们已经从 master 断开的话就使用 ZooKeeper 来判断新的主 master。一旦新的主 Mesos master 被选举出来和 framework 注册进去了，就恢复正常的工作，新的任务就可以调度了。这是在不影响集群中其

他任务正常运行的情况下进行的。

提示：可以在以下网站获得更多 mesos 通过 ZooKeeper 实现领导选举的信息 http://mesos.apache.org/documentation/latest/high-availability。

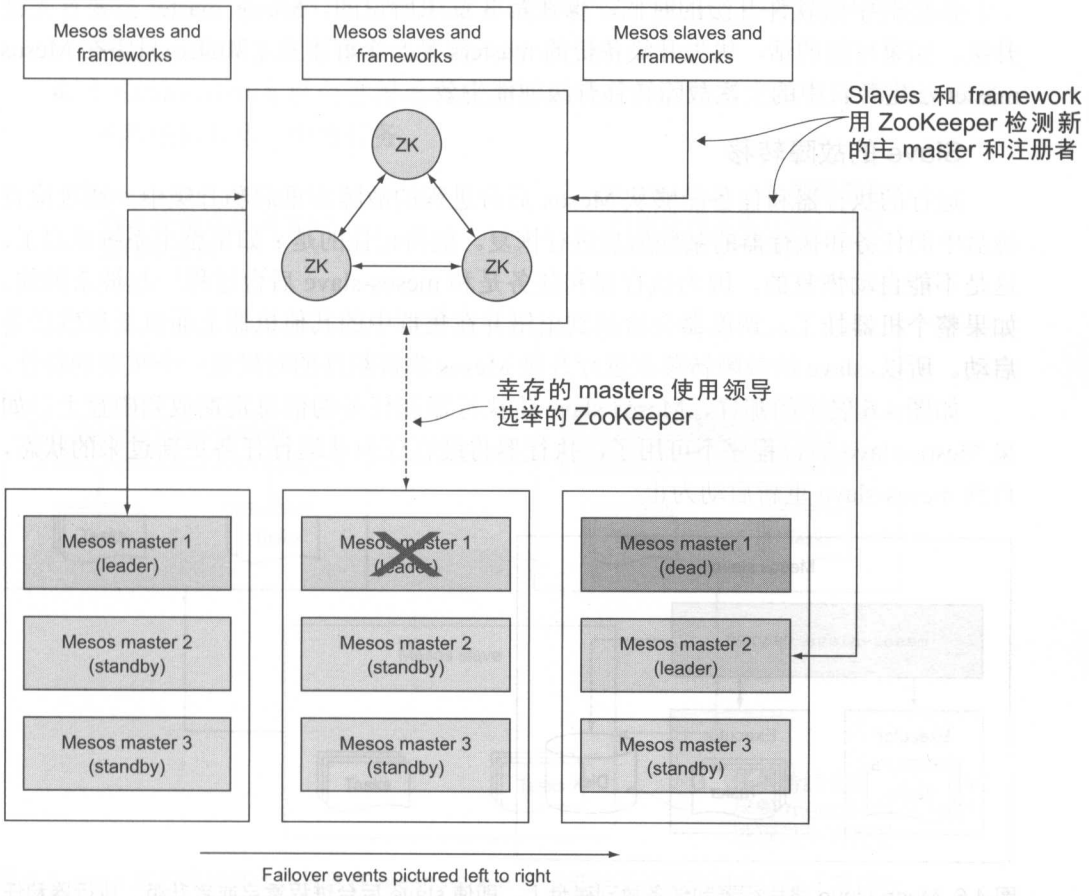


图 4.5 主 master 故障后，剩余的 masters 用 ZooKeeper 选举一个新的领导者。Slaves 和 framework 用 ZooKeeper 检测新的主 master 和注册者

让我们看看图 4.5，它展示了 master 的故障是怎么样转移的。

1. 由之前的选举决定，Mesos master 1 是当前的主 master。Slaves 和 framework 用 ZooKeeper 判断出这个是主 master。
2. Mesos master 1 断线了，slaves 和 framework 检测到它们与 master 断开了，等待新的领导者被选举出来。
3. 领导选举发生了，Mesos master 2 被选举出来做了领导者。记得三个 master

的情况（法定数目是 2），这时不能再有故障发生了，集群当前需要两张选票来选举一个领导者。

4. Mesos slaves 和 framework 检测到新的主 master，注册到新的领导者里面去并恢复正常的工作。它们现在可以得到资源提供了。

在集群中做软件升级的时候，像在第 3 章中所说的，Mesos master 需要首先被升级。如果可能的话，优先升级备份的 masters 是一个好主意；如此，只需在 Mesos master 法定数目中的一次故障转移升级便能生效。

Slave 的故障转移

运行的执行器和任务能够从 Mesos 后台进程的故障、重启和升级中，通过检查磁盘中的任务和执行器的某些信息进行恢复。值得记住的是：如果整个系统重启了，这是不能自动恢复的，因为执行器和任务是和 mesos-slave 后台进程一起被杀掉的。如果整个机器挂了，调度器会检测到出错并在集群中的其他机器上面重新调度任务启动。所以，slave 的故障转移在进行升级 Mesos 集群组件的时候是一个重要的特性。

如图 4.6 展示的那样，Mesos slave 将执行器和任务的信息定期放到磁盘上。如果 Mesos-slave 后台程序不可用了，执行器将缓存任何从运行任务更新过来的状态，直到 mesos-slave 重新启动为止。

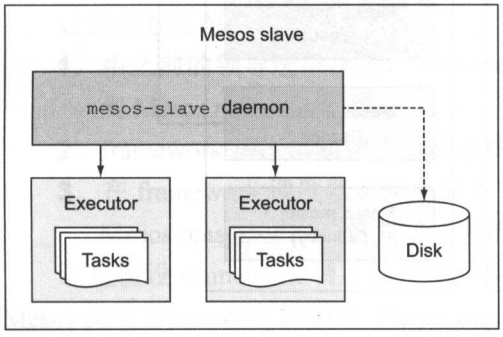


图 4.6 Meso-slave 将执行器和任务放到磁盘上，即使 slave 后台进程重启或者升级，执行器和任务都可以运行

图 4.7 向你说明 mesos-slave 不可用时的几个可能出现的情况，无论是计划中的升级还是进程的异常退出（崩溃）。注意到这些进程都是 slave 默认的行为，它的目标就是尽可能地让任务运行。你可以选择性地配置 --recover=cleanup 这个选项让 mesos slave 杀掉老的执行器进程，这个选项也可能在以后不兼容的升级当中有作用。

让我们看看图中的一些事件：

1. `mesos-slave` 后台进程正常运行着，正常连接着执行器和任务，为 mesos-master 提供正常的升级服务。

2. mesos-slave 断线了，这有很多种原因，进程由于未知的原因挂掉（崩溃）或者正在重新配置或者升级。无论怎样，它失去了与运行中任务的连接。
3. 任务继续持续运行 recovery_timeout 这个选项配置的时间，默认是 15 分钟。如果执行器不能在这个分配的时间内连接上 mesos-slave 后台进程的话，它就会自己退出。当 slave 离线的时候，执行器驱动缓存的状态更新，framework 和执行器本身还是能正常运行的。
4. Mesos-slave 后台进程重新启动，从磁盘上面读取之前保存的状态信息，重新连接到运行中的任务。

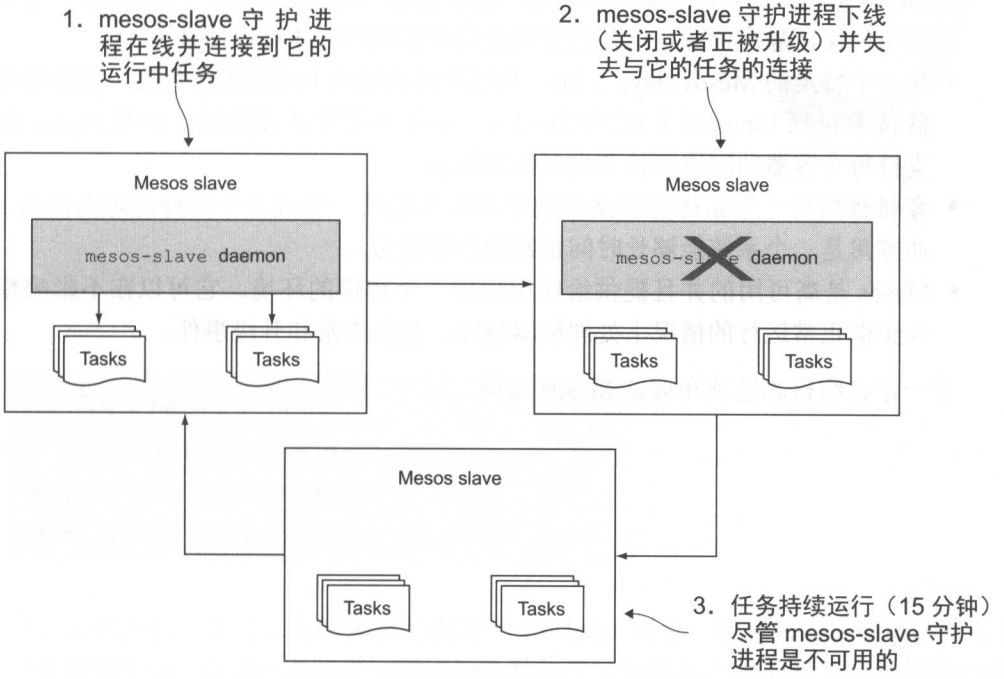


图 4.7 slave 的故障转移允许任务在 mesos-slave 守护进程不可用的时候仍然正常运行

注意：为了让 slave 故障转移，mesos-slave 的后台进程一定要在 75 秒内重新连接到 master。在 Mesos 0.22.2 中，这个是硬编码的超时，不过可以在后面的版本中重新配置（可以查看 https://issues.apache.org/jira/browse/MESOS-2110）。所幸现在的服务系统例如 systemd 和 Upstart 有选项可以自动重启挂掉的进程。另外，你可以使用类似于 Monit 一样的工具来确保服务是一直运行着的，也可以使用类似于 Puppet、Chef 和 Ansible 这样的配置管理工具来管理 slave 的配置、升级和自动重启。

4.4 小结

本章，你在前 3 章获得的知识基础上学习到了 Mesos 是怎样运行的知识。这里有一些事情需要记住：

- Mesos slave 向主 Mesos master 提供资源信息，master 转而向注册在它上面的 framework 提供资源信息。framework 能接收供给列表的全部资源或者一个子集，如果接收的是一个子集，那么剩下的资源就可以提供给其他的 framework。
- Mesos slave 的各个方面可以定制，包括资源、属性和角色。可以为某一特定的角色保留静态资源，但是这样就会导致系统的整体可用性减少。
- 在一个特定的 Mesos slave 上面，利用容器来隔离不同的工作负载。支持的容器技术包括 Linux 的 LXC 和 Docker。尽管不是默认使能的，但是 Mesos 也支持每个容器的网络监控和出口流量限制。
- 容错机制是一个系统能够从容地处理它其中的一个或多个组件故障的能力；高可用是一个系统能够长时间正常运行的能力。
- Mesos 是高可用的并且提供给应用程序一个容错的环境。它可以在不影响整体集群正常运行的情况下处理机器故障、服务失常和升级事件。

下一章介绍日志选项和调试 Mesos 集群。

5 日志记录和调试

本章内容

- 查找日志文件和配置日志记录
- 使用 web 界面调试 Mesos
- 使用命令行调试 Mesos

现在你已经学习了怎样来启动允许一个 Mesos 集群，还有它为应用的运行提供一个容错环境，让我们继续深入调试和看看如何检查集群中服务和工作负载的问题。这一章介绍 Mesos 集群节点和 framework 日志的文件位置和一些日志选项。然后讨论用 Mesos 的 Web 界面和命令行来检查和调试集群的方法和工具。

本章分为两部分：日志记录和调试。第一部分包含不同日志文件介绍，它们的位置和配置的选项。第二部分建立在第一部分的基础上，教会你怎样通过网页界面和命令行工具调试 Mesos 问题和观察任务的输出信息。

我将本章结构化成教程的方式，引领你通过不同的方式从集群中获取信息。我希望你在自己的环境中查找问题的时候会觉得这些方法是有用的。

5.1 理解和配置Mesos日志记录

一个服务产生的日志和服务本身一样重要。很多时候，日志文件没有被用到，它们被归档了，或者放到了日志中心中被遗忘了。但是当问题出现的时候，有用的和详细的日志对于诊断问题是无价之宝。

幸运的是，Mesos 是一个日志记录得非常好的例子。日志文件非常有用并且提供了足够多的信息让系统管理员知道当时发生了什么情况。对于更高级的运维人员和开发者，Mesos 的日志甚至提供了文件名和触发事件被首先输入日志的代码行。

注意：尽管 ZooKeeper 是 Mesos 依赖的一个服务，但还是被认为区分于 Mesos 的。因此本章不包含 ZooKeeper 的日志记录和调试。你可以咨询 ZooKeeper 的管理员指引 http://zookeeper.apache.org/doc/current/zookeeperAdmin.html。

让我们来看看对于不同的 Mesos 集群和 framework 任务的可用日志文件。

5.1.1 日志文件的路径和解释

在管理日志方面，Mesos 为应用管理员提供了很多的灵活性。事件可以被写到磁盘上 Mesos 管理的日志文件或者写到系统日志 (syslog)。由 Mesosphere 的 Mesos 安装包提供的服务脚本也确保了 Mesos master 和 slave 服务的日志被发送到系统日志，这样可以让日志更容易地被类似于 Logstash 或者 Splunk 这样的日志管理服务收集和解析。

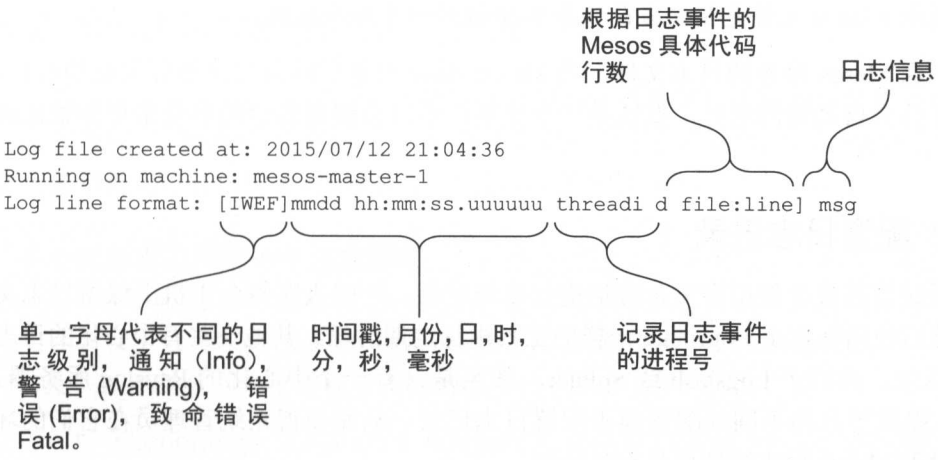
除了服务日志外，Mesos 在每个任务的沙盒里面提供了两个默认日志文件：stdout 和 stderr。这两个特别的日志文件捕获了从控制台输出到标准输出 (stdout) 和标准错误 (stderr) 的日志。这样使得可以在没有登录任务所在机器控制台的情况下看到任务的控制台日志输出。

Mesos 服务的日志文件

Mesos 的 master 和 slave 都是用 --log_dir 这个配置选项来定义 Mesos 服务的日志文件存放在硬盘上的位置。典型的情况下，这个选项一般会设置类似 /var/log/mesos 这样的值，不过也可以被省略让它纪录为标准错误。如果 Mesos 为你管理日志文件，会自动根据大小归档，但是你得确保旧文件是按照一定的格式修剪的。

提示：现在不要担心日志配置的问题，这个问题会在后面的章节中说到。

当 Mesos master 和 slave 服务第一次启动的时候，会输出一些日志文件格式的信息。让我们看看日志文件的格式。



根据上面的日志格式，下面是从一个 Mesos slave 取出来的几行日志：

```
I0713 00:35:04.730430  2217 containerizer.cpp:1123] Executor for container
➥ 'cc538d82-c47b-4b9e-a050-6b6161f658c5' has exited
I0713 00:35:04.730542  2217 containerizer.cpp:918] Destroying container
➥ 'cc538d82-c47b-4b9e-a050-6b6161f658c5'
```

在这两个日志文件例子里面，你可以看到一个任务结束了（容器的执行器退出了），然后 Mesos 就摧毁了没用的容器。每一个例子都包括一个时间戳、进程号和执行这个动作的 Mesos 代码文件和行号。

任务的日志文件

像先前提到的那样，Mesos 在一个任务的工作目录或者沙盒下自动创建两个文件 :stdout 和 stderr。这两个文件放在沙盒中任务需要的其他文件的旁边。你要记得 Mesos slave 在磁盘上的工作目录是使用 --work_dir 这个配置选项配置的，因此，如果你将 Mesos slave 的 work_dir 配置成 /var/lib/mesos 的话，那么指向任务的沙盒的路径就类似于下面这个样子：

```
/var/lib/mesos/slaves
└── <slave-id>/
    └── frameworks/
        └── <framework-id>/
            └── executors/
                └── <task-name>/
                    └── runs/

                        ├── <run-id>/
                        │   ├── stderr
                        │   └── stdout
                        └── latest
```

提示：latest 文件是指向当前任务运行 id 的一个符号连接。

不像 Mesos 服务的日志文件，当 Mesos slave 对老的沙盒目录进行回收的时候，这些文件会自动被清理掉，默认是一个星期，不过会根据磁盘的容量来更频繁地调整。

5.1.2 配置日志记录

系统管理员会使用很多种方法进行日志记录。一些人在每台主机上保留日志文件，然后使用类似于 Logrotate 一样的工具进行定期清理。其他一些有中心化的日志收集系统，类似于 Logstash 或 Splunk，甚至是只有一个中心化的 Rsyslog 服务器。Mesos 提供了几种不同的配置项来配置日志记录，从而使得系统管理员在它们的环境中对于 Mesos 日志管理更具有灵活性。

让我们来看看这些选项，了解一下在什么情况下需要修改它们：

- `log_dir`——将日志文件写到磁盘上一个特定的目录，如果没有指定，不会有日志写到磁盘上，但是事件仍然会写到标准错误里面。这个选项在 master 和 slave 都有，没有默认值。
- `logging_level`——只有在当前（或以上）日志的严重级别才记录日志。可能的值（按照严重程度升序）是，`INFO`、`WARNING` 和 `ERROR`。这个选项在 master 和 slave 上都是有的，默认是 `INFO`。
- `work_dir`——如果在 slave 上面配置了，指定了 framework 工作目录的磁盘位置，这个包含了任务的 stdout 和 stderr 两个文件，默认值是 `/tmp/mesos`。
- `external_log_file`——指明外部管理日志文件的位置，以网页的形式或者 HTTP API 的形式，如果你需要使用日志工具将日志写到操作系统日志里面或者不归 Mesos 管理的日志的话，这将会是很有用的。这个选项在 master 和 slave 上面都可以使用，并且没有默认值。
- `quiet`——当指定的时候，不能记录到标准错误。这个选项在 master 和 slave 都可以用，并且没有默认值。
- `logbufsecs`——在日志信息被刷新到磁盘前被缓存的秒数。这个选项在 master 和 slave 都可以使用，默认是 0（马上刷新日志到磁盘上）。

利用这些选项，无论你选择哪种方式，都可以微调你的 Mesos 日志服务以适应你的日志 framework。以个人来说的话，我宁愿选择 Elasticsearch、Logstash 和 Kibana (ELK) 的形式来收集、解析和索引日志，这样允许你更容易查找和观看日志数据。

现在你已经知道了 Mesos 的日志路径和相关的配置选项，让我们看看一些观察日志输出、查找问题和调试 Mesos 集群的工具。

5.2 调试Mesos集群及其任务

对于分布式系统的调试，尤其是类似于 Mesos 这种可以大规模操作的系统，是非常困难的。一般来说，集群越大，它就越难关联所有节点的集群。作为Mesos的本质，更大的集群可以让资源更好地利用，并且可以避免静态分区。

中心化日志记录的一个注意事项

因为 Mesos 集群是由数十、数百，甚至上千台机器组成的，并且日志文件是存放在不同的集群节点上面的，排查问题可能是一个比较烦琐的过程。幸运的是，可以供日志集中管理与处理的几个选项如下：

- Elasticsearch、Logstash 和 Kibana (也被称为 ELK) ；
- Splunk ；
- Rsyslog 推送到中心服务器。

上面的每一个选项都会在每个主机运行一个小服务用来处理日志文件，并将它们推向中心化的日志 framework。这样可以使你以结构化和可检索的方式在单一的数据中心存储日志，也很容易从一个控制台上面查询和展示日志。

安装这些工具已经超过了本书的范围，不过互联网上面有很多关于这些主题的资源、文档和书籍。

尽管 Mesos 为你调度资源和处理失败，但是有时你需要调试这些失败或者访问集群和工作负载的信息。知道从哪里开始调试和怎样着手下一步是非常有用的，图5.1提供了一个排查问题的工作流程的例子。

在你开始调试 Mesos 之前，要确保 Mesos 服务是运行的，并且确定它们当前的配置。在下面的例子当中，我将 master 和 slave 跑在同一个主机当中，所以两个进程都出现在 ps –ef 和 ps aux 命令的结果当中（根据你传递给 ps 的参数）。配置选项已经略为简洁了，但还是会作为参数出现在 mesos-master 和 mesos-slave 两个可执行文件后面：

```
$ ps -ef | grep mesos
UID          PID  PPID  C STIME TTY          TIME CMD
root        2136     1  0 Jul12 ?        00:00:58 /usr/sbin/mesos-master ...
root        2195     1  4 Jul12 ?        00:14:58 /usr/sbin/mesos-slave  ...
```

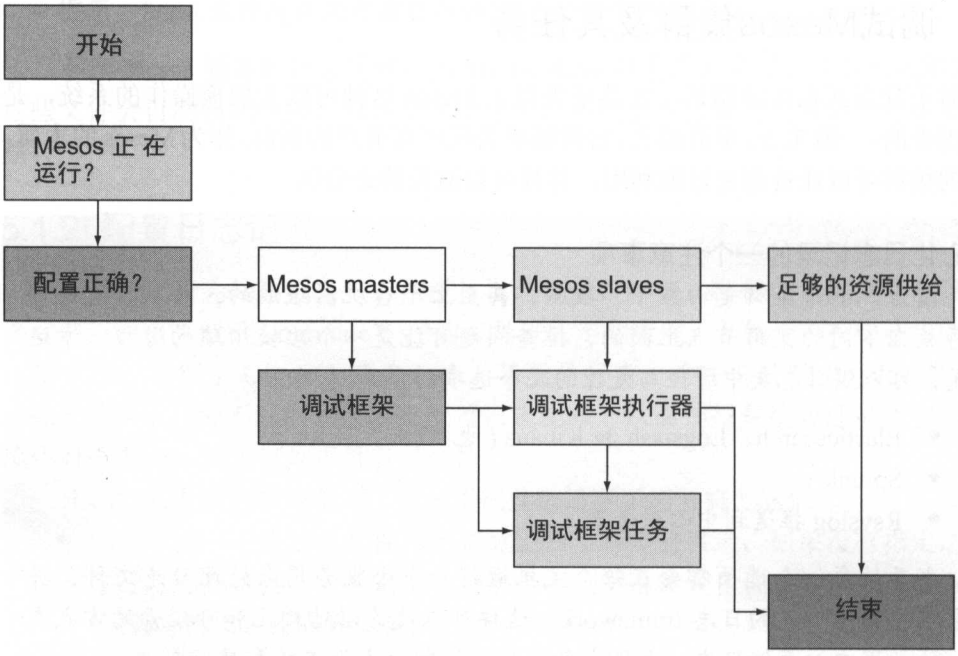


图 5.1 高层次的 Mesos 调试过程概览

如果 mesos-master 和 mesos-slave 服务中间哪一个失败了，查看系统日志看是否出现了问题通常情况下是一个好方法。在 RHEL 上面，这个是在 /var/log/messages；在 Ubuntu 上，系统日志文件是 /var/log/syslog。Mesosphere 的安装包默认把日志输出到标准错误里面，服务启动失败会打印到系统日志中，这样可以把你指引到出问题的地方。

除了确保进程是在预想的配置上面跑起来的，并且观察系统日志里面的服务相关输出，本节的剩余部分还会教给你各种各样的工具和方法来缩小和排查问题。

5.2.1 使用 Mesos Web 接口

Mesos 提供了一个 Web 接口让系统管理员获得集群的状态信息，包括正在运行和已经完成的任务。可以通过访问任意一个 masters 的这个地址 http://mesos-master.example.com:5050 来获取该网页。如果你刚好连到的是非主 master，你会得到一个通知并跳转到当前的主 master。

提示：Mesos 也有一个非常有用的 JSON 格式的 API。你可以访问 master 的 http://mesos-master.example.com:5050/help 或者 slave 的以下地址 http://mesos-slave.example.com:5051/help 来获得更多的信息。第 6 章会提供更多

关于这方面的细节。

Web 接口包括不同标签，让你观察当前集群的状态或者更深入地查看 framework、slaves 和当前运行的任务或已经完成的任务的状态。在下面的几个小节，你会得到一些有用的信息，从而可以快速成功地观察任务的输出和排查问题。

主页

Mesos 页面的 Mesos 主标签展示了活动的和完成了的任务，还有额外的集群信息和数据。这个页面是你基于页面排查问题的开始，从这个页面上，你可以深入看到运行的 framework、任务和 slaves。

在图 5.2 中，你可以看到状态是 RUNNING 的两个任务。其他的任务状态包括：STAGING，FINISHED，FAILED，KILLED 和 LOST，在整个 Web 网站中会呈现。

图 5.2 Mesos 网页主页展示了集群当前的状态

这个特定的页面提供了很多集群的信息。让我们先查看左边的侧边栏，从上面开始：

- 集群总体信息——集群名字，版本，建立日期，服务启动时间，选举时间。
- 日志——打开一个新的窗口展示 Mesos master 的最新日志（假设 log_dir 和 external_log_file 这两个选项都配置上了的话）。
- 集群数据——当前集群数据的快照，包括活动的 slaves 数目、任务的数目（通过其状态来组织）和集群资源（总共的，使用的，已分配的和空闲的 CPU 与内存）。

在主页面中，你可以观察到活动的和已经完成了的任务，还可以看到它们的状态和到它们沙盒的链接。你可以通过点击任务 ID 的链接来调试一个特定的任务。点击沙盒的链接，可以看到容器的内容，包括控制台输出。我会在后面的小节中详细讲述这两个特征。

framework

framework 的标签列举出了活动中的和已经终止了的 framework，还有它们所分配和已经使用的资源。在这个页面上，你也可以看到 framework 注册的日期和时间，或者其注册时是否断开了连接或故障恢复。

提示：通过点击页面上的相对时间（例如 2 分钟前），页面就会展示实际的时间点。

例如在图 5.3 中，你可以看到 Marathon framework 用了 0.1CPUs、128MB 内存和一个活动的任务。

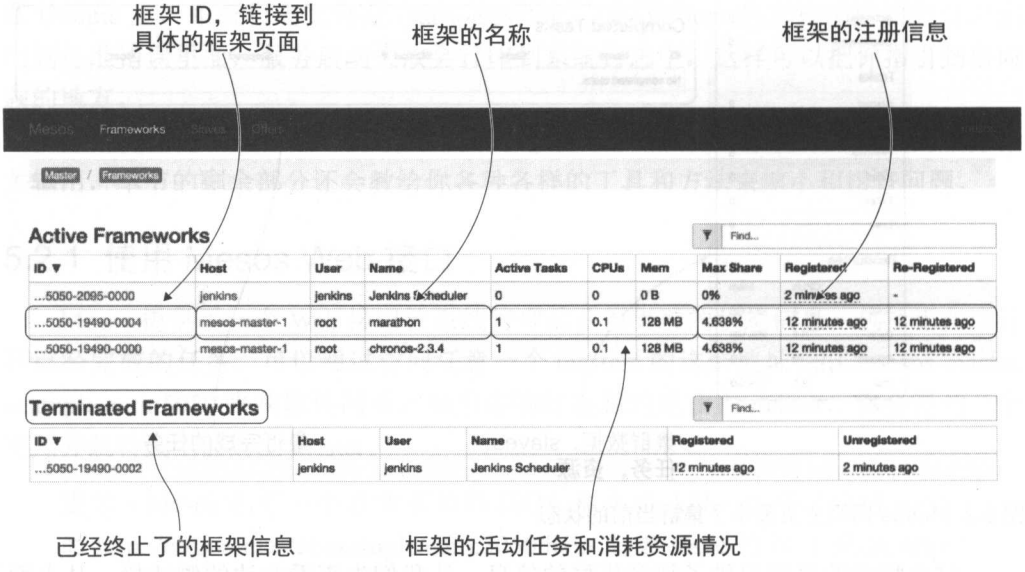


图 5.3 framework 页面展示注册和终止了的 frameworks，也包括 framework 活动的任务数、资源消耗的情况，还有 framework 注册的日期和时间

点击特定 framework ID 的链接会引导你去另外一个展示 framework 任务的页面，这里还展示了任务的当前和最后的状态信息。如果 framework 注册了一个 URL，点击 Hosts 那一列会引导你到相关的应用程序 Wed 接口页面。

任务

在浏览到 framework 级别页面的时候，你可以观察到 framework 当前活动的和已经终止了的任务，还有 framework 的信息例如名字、用户和页面。图 5.4 展示了一个例子：

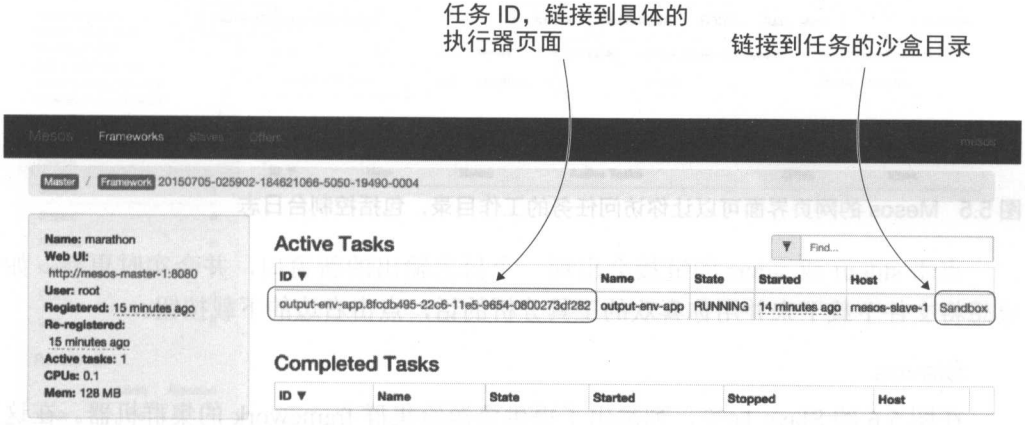


图 5.4 特定 framework 的活动和完成了的任务

为了调试 framework 的一个特定任务，可以点击包含任务 ID 的链接。有关任务的信息会出现在左边的侧栏，包含执行器的名字、消耗的 CPU 和内存资源信息。

每一个任务都在一个沙盒的环境或者专门的工作目录中运行，但是仍然可以访问主机系统上面的工具和库，但是有一个显著的例外是：运行在 Docker 容器里面的进程，只能访问容器里面的工具和库。然而，当 Mesos 启动一个 Docker 容器时，它自动将任务的沙盒工作目录挂载到容器里面，以方便使用类似于 Mesos 提取器这样的功能从沙盒下载相关的文件。

在 Mesos 的网页界面上，每一个任务的沙盒链接都打开一个像图 5.5 展示的那样的图形文件浏览窗口。

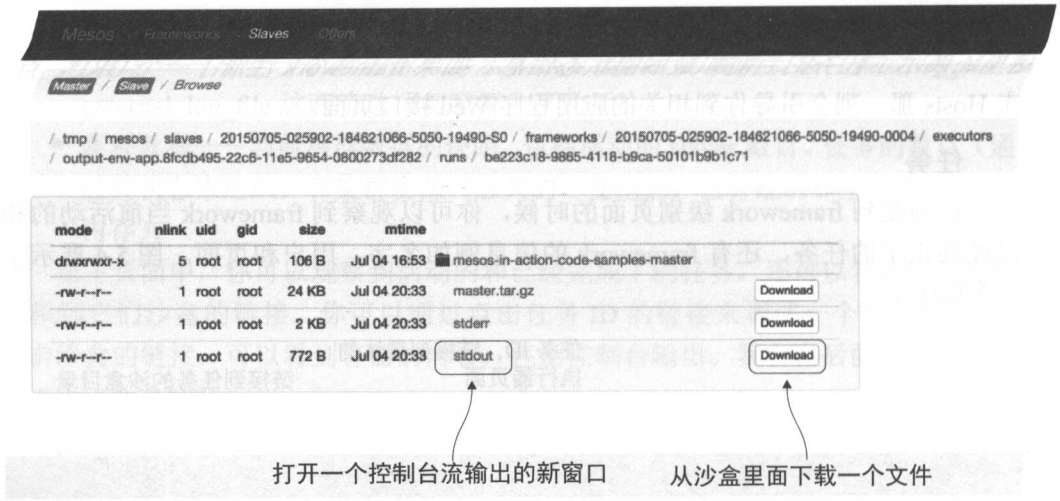


图 5.5 Mesos 的网页界面可以让你访问任务的工作目录，包括控制台日志

点击 stdout 或 stderr 的链接会出现一个日志输出的新窗口，并会实时更新。如果想将文件下载下来并用你喜欢的工具分析的话，点击右边的下载按钮。

Slaves

在图 5.6 的 Slave 标签，列举出了提供资源给集群 framework 的集群机器。在这个页面你可以看到注册到集群的 slaves 和它们组合的资源。

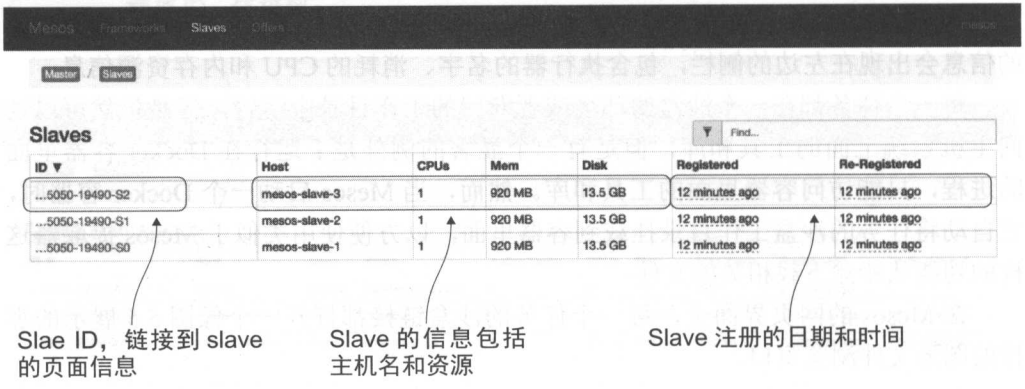


图 5.6 slave 页面列举了注册进来的 Mesos slave 信息，包括 slave ID、主机名、资源信息和注册的日期和时间

点击 slave 的 ID 会向你展示类似于图 5.7 一样的页面，你可以看到当前所有 framework 在这个 slave 上面使用的资源情况。这对于调试问题或观察在 slave 上面的一个活动是非常有用的，并可以确定集群中的哪个机器在哪段时间做了什么。

图 5.7 特定 Mesos slave 的状态页面，信息包括活动的 framework、任务和资源

在这个例子里面，使用这个 slave 的唯一 framework 是 Marathon。如果你点击 Marathon 的 framework ID 的链接，你可以看到 framework 启动的执行器信息。再深入一层的话，你可以看到正在运行的任务、它们的状态和资源的消耗情况（如图 5.8 所示）。

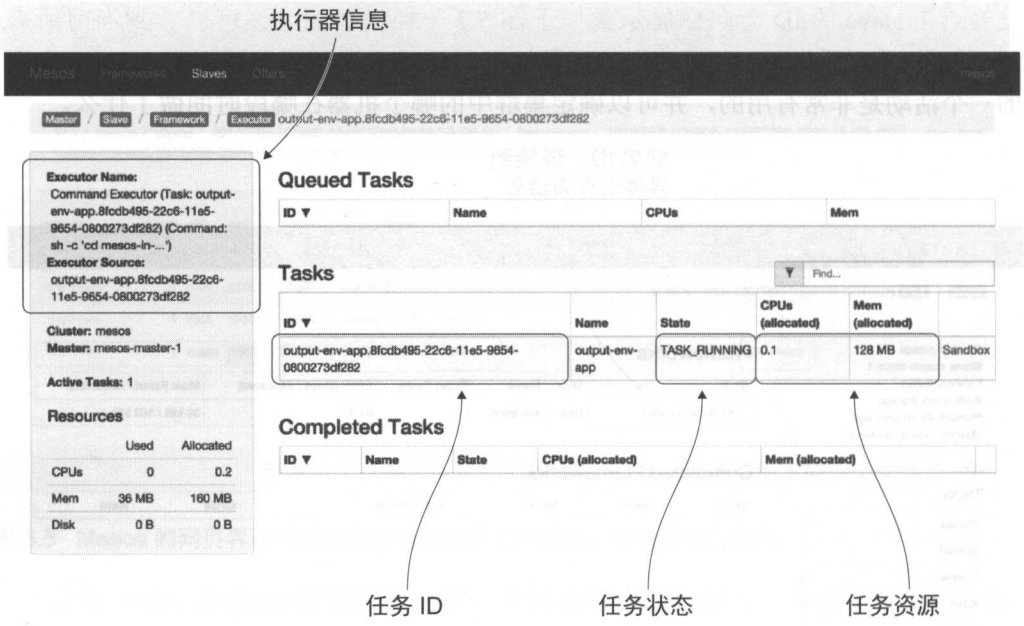


图 5.8 特定 slave 下面特定 framework 的执行器和任务详细信息

供给（OFFERS）

供给选项卡列举出了集群中未完成的资源供给，由 Mesos master 提供给 framework 的资源中并没被 framework 接收或者被 framework 拒绝的资源供给会展示在这里。你可以查看图 5.9 中不良的调度例子，提供的一些资源没有被 framework 接收或被 framework 拒绝了。

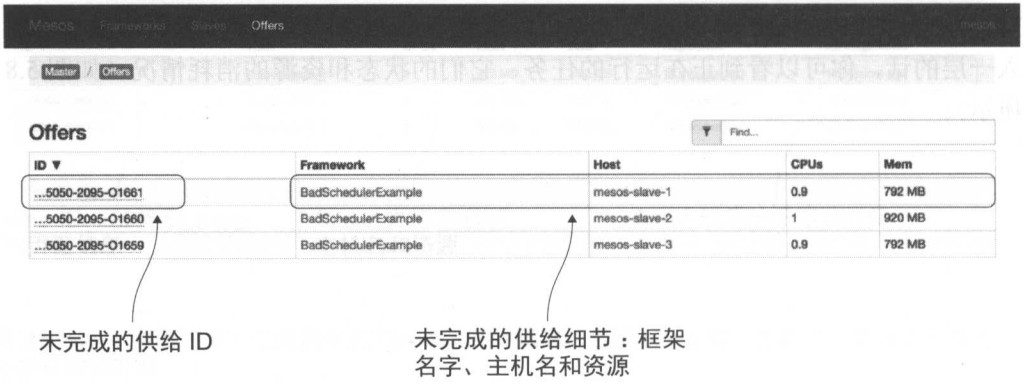


图 5.9 供给页面列举出没有被接收或被拒绝的未完成资源

总体来说，你需要用来观察集群状态的所有信息在页面中都是可以找到的。但

是对于那些喜欢用命令行工作的朋友来说，Mesos 也有命令行的工具来提供和页面同样多的信息。现在你既然已经学习到了用页面的形式来调试一个 Mesos 集群和 framework 的一般过程，不妨也来学习一下 Mesos 绑定的一些内建命令行工具。这些工具使得你用了同样的处理过程，不过不是页面的形式而是命令行。

5.2.2 使用内置命令行工具

除了 Web 接口和 HTTP APIMesos 外，Mesos 还为那些喜欢用命令行工作的朋友提供了内建的命令工具来替代页面浏览器。值得注意的是本节提到的一些工具仅仅在安装了 Mesos 的系统中才会有。

概览

类似于它们 GNU 的核心工具集（Coreutils），Mesos 相应地提供了一个小的命令行工具集用来管理集群和处理文件与进程。这些工具包括如下：

- `mesos cat`——为一个任务连接和打印一个文件。
- `mesos execute`——启动集群的一次性命令。
- `mesos ps`——类似于 Linux 和 UNIX 的工具 ps，这个命令输出一个集群中运行命令的一个列表。
- `mesos resolve`——查询 master 或 ZooKeeper 来确定当前的主 master。
- `mesos scp`——将一个本地文件复制到 Mesos masters 所知道的 slaves 的远程目录上面。
- *`mesos tail`——类似于 Linux 的 tail 命令，将特定任务的一个文件的最后 10 行打印出来。*

要了解完整的可用命令，执行 mesos help 命令。

例子

为了更好地说明这些命令行工具，我想最好是介绍一些现实世界的例子。对于下面的每一个命令，ZooKeeper 和 Mesos master 都在同一个主机上面运行：mesos.example.com。注意，为了简便我省略了每个命令的输出结果。

输出集群中当前运行的命令：

```
$ mesos ps --master=zk://mesos.example.com:2181/mesos
```

确定当前的主 master：

```
$ mesos resolve zk://mesos.example.com:2181/mesos
```

在集群中运行一次性命令，记下任务运行后输出的 framework ID，下一个例子

将会用到：

```
$ mesos execute --command="echo 'Hello, Mesos'" --name=HelloMesos
➥ --master=$(mesos resolve zk://mesos.example.com:2181/mesos)
```

最后，观察你刚才运行命令的标准输出，执行下面的命令。确保用刚才命令的framework ID 替换—framework 变量的值：

```
$ mesos cat --master=$(mesos resolve zk://mesos.example.com:2181/mesos)
➥ --framework=20150712-210436-16842879-5050-2136-0002
➥ --task=HelloMesos --file=stdout
```

如果你选择在控制台工作的话，这些内建的命令行工具是非常有用的。但是有一个限制就是必须在你的系统中安装 Mesos。现实当中，许多系统管理员喜欢在自己的手提电脑或者工作站当中调试问题，不愿意登录到生产环境运行一些调试工具，或者在本地安装 Mesos。

幸运的是，Mesosphere 的工作者们开发了他们自己的 Mesos 命令行来和远程的 Mesos 交互。这些工具使得你在不安装 Mesos 的情况下能够运行目前为止学到的命令。让我们来看一下。

5.2.3 使用 Mesosphere 的 mesos-cli 工具

Mesosphere 的团队为 Mesos 开发了一个基于 Python 的，集群范围内的调试工具，并巧妙地命名为 mesos-cli。它可以让你在不安装 Mesos 的情况下，从你的手提电脑或工作站向 Mesos 集群执行命令。这个工具以更友好的形式重新实施前面提到的命令行工具。

mesos-cli 在 Python 包索引（PyPI）里面可用，你可以在你的工作站运行下面的命令来安装它：

```
$ sudo pip install mesos.cli
```

注意：如果 Python 的 `pip` 工具在你的系统中没有安装，请参考 https://pip.pypa.io/en/stable/installing/ 里面的指令来安装它。

安装完成后，可以看到一个新的叫 /usr/local/bin/mesos 的可执行文件。只要 /usr/local/bin 这个路径是在你的环境变量 `$PATH` 中出现了，mesos-cli 的可执行文件 mesos 就取代了 `Mesos` 内置命令行工具的位置了。现在我们来看看 mesos-cli 里面的命令。

概览

类似于它们的核心工具集和 Mesos 自身绑定的工具，在这里我包含了一些热门的 mesos-cli 的子命令：

- `mesos cat`——为一个任务连接和打印一个文件。
- `mesos config`——在命令行配置 mesos-cli 工具。
- `mesos events`——按顺序地从集群节点中提出 master 和 slave 的实时日志。
- `mesos find`——通过给出的任务 ID，找到并列举出沙盒中的文件。
- `mesos head`——类似于 Linux 和 UNIX 的命令，输出文件的前几行。
- `mesos ls`——类似于 mesos find，列举出任务的沙盒中的所有文件。
- `mesos ps`——类似于 Linux 和 UNIX 的命令，输出集群中运行的任务信息列表。如果使用 -i 参数可以输出非活动的任务。
- `mesos resolve`——查询 master 或 ZooKeeper 来确定当前的主 master。
- `mesos scp`——将一个本地文件复制到 Mesos masters 所知道的 slaves 的远程目录上面。
- `mesos ssh` ——通过给出的任务 ID，允许你直接 SSH 到特定任务的沙盒。这个命令假设你有权限 SSH 到你的集群节点。
- `mesos stat`——以 JSON 的格式输出集群中的 master 或者 slave 的信息。
- `mesos tail`——类似于 Linux 和 UNIX 的命令，输出文件的最后几行。

提示：如果你希望这些命令在你的 shell 外壳里面通过按 tab 按键完成，那你走运了，将下面的命令添加到 ~/.bash_profile 当中：`complete -C mesos-completion mesos`。

为了查看 mesos-cli 里面的完整命令，你可以执行 mesos help 命令。

例子

为了更好地说明这些命令行工具，我想最好介绍一些现实世界的例子。对于下面的每一个命令，ZooKeeper 和 Mesos 都跑在同一个主机上面：mesos.example.com。注意，为了简便我省略了每个命令的输出结果。

提示：你可以通过执行 mesos <subcommand> --help 获得子命令的帮助。

配置 mesos-cli 用来确定当前主 master 的 URL：

```
$ mesos config master zk://mesos.example.com:2181/mesos
```

输出当前集群运行着的任务：

```
$ mesos ps
```

在 Docker 镜像里面执行一条一次性的命令：

```
$ mesos execute --command="cat /etc/redhat-release" --name="releaseTask"
➥ --master=$(mesos resolve) --docker_image="centos:7"
```

最后，观察你上一条命令的标准输出：

```
$ mesos cat releaseTask stdout
```

像在这里阐述的一样，由 Mesosphere 提供的 mesos-cli 工具是基于命令构建的，允许你从你的个人工作站舒适安全地和 Mesos 集群与工作负载进行交互，甚至对它们调试问题。由于它是基于 Python 的，并且是由 Mesos 基础代码之外维护的，它也可以很轻松地根据你的需求进行定制。

5.3 小结

本章你学习到了调试 Mesos 集群的不同方法和工具。这些包括日志文件位置、日志配置选项、Mesos 网页界面接口和不同的命令行工具。需要记住的一些事情如下：

- Mesos 对于 master 和 slave 都提供了各种不同的日志配置选项。你可以配置日志路径，日志级别，还可以拥有外部管理的日志。
- 每一个 Mesos 的任务都有两个文件：stdout 和 stderr。这两个文件存放在任务的沙盒里面，分别获取任务的标准输出和标准输出日志信息。
- 在你进行调试问题之前，你要确保 mesos-master 和 mesos-slave 服务运行着你想要的配置选项。可以通过观察 ps aux | grep mesos 的输出来确定这些。
- Mesos 提供了一个网页接口和 HTTP API 来查询集群的状态和协助问题调试。你可以通过 http://mesos-master.example.com:5050 这个地址访问页面。如果你访问的不是当前的主 master 的话，会重定向到当前的主 master。第 6 章会介绍 API。
- Mesos 有内置的命令行工具协助调试和问题排查。Mesosphere 同样也提供了一个基于 Python 的工具叫 mesos-cli，可以从 PyPI 下载。

第 6 章将完成本书第 2 部分介绍，涉及生产环境中运行 Mesos 的信息。它的主题包括：监控、安全、以及增加、移除和替换 masters。

6 生产环境中的Mesos

本章内容

- 监控 Mesos masters、slaves 和 ZooKeeper 集群
- Mesos REST API 导航
- 增加、替换和移除 Mesos masters
- 理解和配置认证、授权和流量限制

值得庆祝的是，现在已经到达了本书第 2 部分的最后一章了。到这个点上，你已经学会了 Mesos 是怎样提供一种方法来提高数据中心的效率的；怎样配置和安装 Mesos、ZooKeeper 和 Docker 的；Mesos 是怎样提供一个容错环境给应用运行的；怎样和一个运行的 Mesos 集群交互，以及怎样调试它的。

本章——可能也是本书最重要的章节之一——包括了很多实践的东西：监控的细节、法定数目的改变和访问控制的规则，这些你都需要知道，以方便你在生产环境中做出微调和运行应用程序。本章的内容也很好地为第 3 部分做了准备，第 3 部分会介绍用 Marathon、Chronos 和 Aurora 的 framework 在集群中来启动长时间运行的应用程序和调度任务。

6.1 监控Mesos和ZooKeeper集群

像你之前学到的一样，Mesos 有两种主要的服务：`mesos-master` 和 `mesos-slave`。在最基本的层次上，你可以配置一个监控系统来确保这些进程在 Mesos 的集群系统中是启动和运行着的，但我们所有人都知道这样层次的监控是不满足需求的。幸运的是，Mesos 提供了丰富的基于 JSON 格式的 HTTP API 接口供你查询关于集群健康状态的更多信息。

> **提示**：如果你想探索从 Mesos API 命令行输出的 JSON 格式串的话，你可以考虑使用 HTTPie 这个友好的，类似 cURL 的工具，它可以格式化和为输出增色。更多的信息可以访问：https://github.com/jkbrzt/httpie。

考虑到 Mesos 项目正在快速地发展当中，在本章中覆盖到 API 的每一个细节是不实际的。下面的小节包含了最常见的——可能也是最重要的——需要监控 Mesos 和 ZooKeeper 集群的知识。你也可以浏览下面的网页以获得更多关于特定 Mesos 版本的知识：

- http://mesos-master.example.com:5050/help
- http://mesos-slave.example.com:5051/help

更多的关于 API 的参考资料可以在 Mesos 网上监控文档找到：http://mesos.apache.org/documentation/latest/monitoring。

6.1.1 监控 Mesos master

监控机器的数目满足 Mesos master 的法定数目是确保集群能提供你的用户所期望的那种级别服务的关键，这样新的任务就能确保在这些组成集群的机器上被调度起来。在很多情况下，这需要远高于基本主机监控级别的监控和指标 (CPU，内存，磁盘，网络) 和进程监控（`Mesos master` 的服务）。

尽管有很多的监控工具，但本节将范围缩小，用 Nagios，一个很热门的开源监控平台，来监控 Mesos master。Nagios 可以通过查询 Mesos REST API 可用的指标，这对于你开发自己的监控检查是非常有用的。

转发 HTTP 请求到主 Mesos master

如你在本书之前学到的那样，连接到一个非主 Mesos master 的网页接口会自动重定向到主 master。但是如果你去查询非主 master 的 API 又会怎么样呢？对于 Mesos 0.22，情况有点不可预知；你可能会从 API 中得到不正确或不完整的数据。

为了总能确保你或者你的监控系统从主 master 里面获得准确的数据，为了监控和管理的目的你可以考虑将一个 HAProxy 实例放到你的 Mesos master 前面。

下面这个从 HAProxy 的配置文件里面提取的摘录将每一个 Mesos master 添加到了负载均衡池里面。同时添加了一个健康检查，这个只有主 master 才会成功。通过这种方法，就可以确保你的请求转发到主 master 上面了，是三个主机的其中之一。

```
listen mesos-master 0.0.0.0:5050
  mode        http
  option      httpclose
  option      forwardfor
  option      httpchk GET /metrics/snapshot
  http-check expect string "master\/elected":1
  server      mesos-master-1 mesos-master-1.example.com:5050 check
  server      mesos-master-2 mesos-master-2.example.com:5050 check
  server      mesos-master-3 mesos-master-3.example.com:5050 check
```

同时为了确保在你的基础架构中 HAProxy 不是一个单一出错点，你可能需要通过主备配置的模式部署 HAProxy，在两个实例之间运用 keepalived 或者一个浮动的 IP 地址来做高可用架构。另外如果你不想在你的 masters 前面部署一个 HAProxy 实例，可以查询 /master/redirect API 路径，这个路径会返回一个 HTTP 307，重定向到主 master 页面。

用 Nagios 监控 Mesos master

Nagios，一个开源的，通过实战测试的监控系统，在数据中心和服务监控中是无处不在的。因此似乎讨论用 Nagios 监控 Mesos 是一件自然而然的事了。

一个叫 OpenTable 的团队已经为 Mesos 的监控创建了一个 Nagios 检查点。你可以从这里下载脚本：https://github.com/opentable/nagiosmesos。它有能力监控主 master 的以下几个情形：

- 基本健康检查；
- 最小数量注册的 slaves；
- 注册的 framework。

要查看完整的选项列表，可以执行 check_mesos.py –help。另外，下面的例子是使用 check_mesos.py 的最基本形式：

```
./check_mesos.py --host mesos-master.example.com
```

如果你想得到更多的集群信息或者开发你自己的监控脚本的话，就得查找 Mesos REST API 了。你会在下一小节浏览这些 API 路径和它们返回的数据。

查询 Mesos master API 路径

Mesos 提供了一个广泛的 REST API 来获取集群信息。这些信息包括但是不限于下面的指标：

- 利用的资源；
- 接收并处理的 framework 消息（如果启用了身份验证）；
- 系统负载；
- 连接上来的 slaves。

由于项目在快速发展当中，列举出所有的 API 路径是不现实的。表 6.1 里面包含了一些最重要的 API：

表 6.1 选出来的一些 Mesos master API 路径

路　径	描　述
/help	返回可用的 API 路径
/metrics/snapshot	从 master 返回一个包含系统指标的 JSON 对象
/master/health	如果 master 是健康正常的话返回 HTTP 200(OK)
/master/redirect	返回一个 HTTP 307(暂时性重定向) 重定向到主 master 的 URL
/master/slaves	返回连接上来的 Mesos slave 信息

为了得到更广泛的路径列表，你可以访问 /help 这个路径，或者咨询你特定版本的 Mesos 官方文档。

6.1.2 监控 Mesos slave

监控 Mesos slave 可能比监控 masters 的重要性小一点，因为 slave 并不需要维持一个运行的法定数目和并不需要在集群中对于在哪里调度任务作出决定。然而在生产中，监控这些工作机器和监控其他机器是一样重要的。如果没有适当的监控 slave 的方法，你就会在资源耗光或磁盘用满的情况下甚至得不到一个警告的风险。

这里没有一些固定的指引，任何一个组织或者环境都有自己对于 CPU、内存和磁盘的阈值。无论怎样，这里有一些对于 Mesos slave 的监控建议：

- 确保 `mesos-slave` 进程是运行的（端口 5051 是可以访问的）；
- 确保 docker 或者 docker.io 进程是运行着的（如果你使用 Docker 的话）；
- 监控基本的 CPU、内存、磁盘和网络，定期地收集数据和绘制成图；
- 监控每一个容器的指标（CPU，内存，磁盘，网络）。

除了每个容器的监控（本章后面会讲到）之外，其他的都是基本的操作系统级别的监控了。这些指标也有一些暴露到 Mesos slave 的 API 路径指标里面了。现在我们讲述对你和你的监控系统有用的 Mesos slave 的 REST API 信息。

查询 Mesos slave API 路径

就像 Mesos master 一样，Mesos slave 也有一个广泛的 REST API 给你查询。但是不同于 master 返回整个集群范围的指标，slave 的 API 路径返回的是某一个特定 slave 的信息。由于项目在快速发展当中，列举出所有的 API 路径是不现实的。表 6.2 包含了一些 API 路径，在我看来对本小节来说是很重要的。

表 6.2 选出来的一些 Mesos slave API 路径

端　点	描　述
/help	返回可用的 API 端点
/metrics/snapshot	从 slave 返回一个包含系统指标的 JSON 对象
/monitor/statistics.json	返回跑在 slave 上面的容器的资源消耗情况的一个 JSON 对象

这些路径能够提供你期望的操作系统级别的资源指标，再加上一个特定 Mesos slave 上面的容器的指标。为了得到更广泛的路径列表，你可以访问 /help 这个路径，或者咨询你某个版本的 Mesos 官方文档。

6.1.3 监控 ZooKeeper

尽管本书主要是关于 Mesos 的，但是 ZooKeeper 对于管理协调和发现是至关重要的。因此，为了有一个稳定的 Mesos 部署环境，确保 ZooKeeper 被充分地监控是很重要的。尽管在前面的章节向你指出了网上 ZooKeeper 文档的地方，但是我想在本节最好向你指出监控 ZooKeeper 的正确方向。

注意：本节的剩余部分是假设你使用 ZooKeeper 3.4.0 以上的版本。

通过 Exhibitor 管理和监控 ZooKeeper

为了监控和管理 ZooKeeper 的安装，一个 Netflix 的团队开发和开源了一个他们命名为 Exhibitor 的工具。引用项目的页面来说，Exhibitor 是一个 Apache ZooKeeper 的监控系统。它提供了系统管理员集群范围内的管理和监控功能，如下：

- 确保一个实例是运行和回应请求的；
- 执行备份和恢复；
- 管理集群范围的配置；
- 通过 Web 接口和 REST API 探索一个 Znodes 的树。

要得到 Exhibitor 的更多信息，包括安装配置的指令，浏览项目的网页：https://github.com/Netflix/exhibitor。

通过 Nagios 监控 ZooKeeper

除了 Exhibitor 之外，各种不同的监控脚本都使用 ZooKeeper 来发布。如果你是从源代码安装的 ZooKeeper，或者从源代码编译的 Mesos 并使用了绑定版本的 ZooKeeper 的话，这些监控脚本可以在 src/contrib/monitoring/ 这个目录找到。因为我们已经讲到了用 Nagios 监控 Mesos，同样我们也要讲到用 Nagios 监控 ZooKeeper。

> **提示**：如果你通过管理包的方式安装 ZooKeeper 的话，是没有这些监控脚本的。你可以到如下的网址去下载 ZooKeeper 的 Nagios 监控脚本 https://github.com/apache/zookeeper/tree/trunk/src/contrib/monitoring。

一个 Nagios 的 ZooKeeper 监控脚本，check_zookeeper.py，能在下面这个目录找到：src/contrib/monitoring/check_zookeeper.py。这个脚本会有几个参数，当某个指标超出特定的范围的时候会向你示警。Nagios 服务配置的例子也是在 nagios/ 目录下面可以找到，可以用来配置你的 Nagios 服务器。

在下面的例子中，我会直接调用 check_zookeeper.py 这个脚本来说明它是怎样工作的。它会检查 ZooKeeper 节点还未完成的请求，如果这个数目达到 10 的话就会有警告信息，达到了 25 的话就会有紧急情况发生：

```
$ python check_zookeeper.py -o nagios -s zk.example.com:2181
➥ -k zk_outstanding_requests -w 10 -c 25
```

这是众多监控 ZooKeeper 健康状态例子中的一个。除了 nagios/ 目录下提供的服务检查的例子之外，你也可以通过执行下面的命令得到用法上的全部帮助信息：

```
$ python check_zookeeper.py --help
```

你也可以通过一些内置的查询 ZooKeeper 的命令来写你自己的 Nagios 监控脚本，后面我会为你讲解一些。

通过四字命令监控 ZooKeeper

可以通过向 ZooKeeper 发送一些四字母的命令来获得某一个实例的一些信息。表 6.3 提供了一些选择出来的对监控有用的四字命令的细节：

表 6.3 选出来的监控 ZooKeeper 的命令

命 令	描 述
ruok	基本的健康检查；如果服务是启动正常的话返回 imok
mntr	返回一个用来监控 ZooKeeper 实例的制表符分隔的指标。这些信息包括服务器的状态（standalone，leader，follower）、版本和延时情况
srvr	提供服务器细节信息，包括连接的个数和模式（standalone，leader，follower）

续表

命 令	描 述
stat	返回连接的客户端列表和服务器状态的细节

四字命令通常是通过 Netcat 工具发送到 ZooKeeper 实例的。下面的命令通过 netcat 向监听 2181 端口的一个 ZooKeeper 实例发送 ruok 的健康检查命令：

```
$ echo 'ruok' | nc zk1.example.com 2181
imok
```

提示：可以在下面地址查看 ZooKeeper 管理员手册来查看完成的 ZooKeeper 命令列表 http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_zkCommands。

6.2 修改Mesos master的法定数目

有时候你会发现需要修改集群中 Mesos masters 的数目。这可能是由于你需要更换故障的硬件，在一个新发布的操作系统上重新构建虚拟机，或者增加额外的 masters 来提高你集群的高可用策略。本小节涵盖了在运行的部署环境中修改 Mesos masters 的数量的知识。

谨慎修改 masters 的法定数目

尽管本小节提供了修改 masters 数量和 masters 法定数目的指令，但是花费一些时间来注意几个事项是值得的。

Mesos 的复制日志没办法做到零停机时间的再配置，因此为了通过 -quorum 的配置选项来改变集群的法定数目的话，masters 就必须得重启。你可以在本小节后面学到如何执行这个操作。为了获得未来 Mesos 发行版本关于配置的更多信息，可以访问如下的网站 https://issues.apache.org/jira/browse/MESOS-683。

另外，Mesos 没有执行一份参加法定数目的 masters 白名单。为了防止复制日志的崩溃和潜在性的脑裂行为，你需要确保加入法定数目的 masters 数量不能超过相应的法定数目（就像表 6.4 展示的那样）。对于 Mesos 0.22.x，有一个没有完成的支持白名单的功能需求。更多的信息可以查阅 https://issues.apache.org/jira/browse/MESOS-1546。为了避免这个风险，你可以开启你 Mesos 使用的 ZooKeeper 集群的认证功能来阻止没有认证的 masters 加入法定数目。

未能遵循这些基本规则可能会导致服务中断。像所有的生产服务一样，要在准生产环境测试变更，还有就是在生产环境变更的时候需要谨慎。

正如第 3 章提到的，维持一个法定数目需要奇数个的 master 数目；集群中大部分的 maters 需要作出决定。法定数目是在每一个 masters 中使用 --quorum 的选项配置。为了方便参考，在这里重复使用第 3 章中法定数目的表格做了表 6.4。

表 6.4 Mesos masters 的法定数目

master 数目	法定数目	可容忍的失败机器数
1	1	0
3	2	1
5	3	2
2*N*-1	*N*	*N*-1

现在你有了一些关于修改法定数目大小的背景信息（还有几个注意事项），让我们看看怎样在一个集群增加 masters。下面每一个小节都是一个例子场景，为每一个步骤提供了指令。

6.2.1 添加 master 节点

有时候你会想增加 Mesos master 的数目来获得额外的故障恢复能力，例如，你可能想要能够忍受两台 masters 主机故障而不是一台。在这个场景下，我会说明怎么将 masters 从 3 增加到 5，这样就允许你的集群可以忍受 2 个 masters 的故障了：

1. 当前你有 3 台 Mesos masters 运行着配置 `--quorum=2`。重新配置每个 masters 为 `--quorum=3`，并且通过运行 `sudo service mesos-master restart` 重新启动每一个 `mesos-master` 服务。
2. 增加两个 masters 也是配置 `--quorum=3`，并且也是通过 `sudo service mesos-master restart` 来重启 `mesos-master` 服务。

提示：如果 ZooKeeper 是和 mesos masters 在同一台机器上运行的话，现在也是一个增加 ZooKeeper 集群大小的最好时机。大体上来讲，这包括配置新的节点加入集群，然后重新配置追随者（followers），再然后重新配置和重启当前的 leader。还有就是不要忘记更新 Mesos masters、slaves 和 frameworks 用到的 ZooKeeper URL，增加刚加上来的新成员。

6.2.2 移除 master 节点

尽管很少有人会减少法定数目，但是还是可能发生的。在下面的场景中，我会减少集群 masters 的数量，由 5 到 3。这样集群能够容忍失败的 masters 就由 2 减少到 1：

1. 当前你有 5 台 Mesos masters 运行着配置 `--quorum=3`，从集群中移除两台 masters，并且确保它们不会再启动了。
2. 重新配置剩下的 3 台 masters 为 `--quorum=2`，并且也是通过 `sudo service mesos-master restart` 来重启 `mesos-master` 服务。

6.2.3 替换 master 节点

如果需求升级需要替换其中一个 masters 的话（移除一个 master 并在它的位置上面增加一个 master 以保持集群的法定数目），你应该可以在不需要任何重新配置或停机的情况下做到这一点。在这个情况下，你会退役一个老的 master 并增加一个新的：

1. 当前你有 3 台 Mesos masters 运行着配置 `--quorum=2`。移除那台你想替换掉的 master，并且确保它不会再加入到集群中（如删除虚拟机，格式化磁盘诸如此类的）。
2. 增加一台 master，并且配置它和剩下的 master 一样的 quorum，在这个例子中是 `--quorum=2`。
3. 通过运行 `sudo service mesos-master restart` 来重启 `mesos-master` 服务，并且让复制日志跟上现存的 masters。

6.3 安全和权限控制的实施

一个良好的安全战略对于一个组织的基础架构安全性是至关重要的，尤其是最终的数据。今天我们看到一个 *security-in-layers* 的方法运用得最好，给了系统一些主意——例如基本用户认证，另外也结合了一些更复杂的，首要的想法——例如网络分区，还有就是主机和数据中心之间的加密。

Mesos 实施的安全模型能被区分为两个你们早已经熟知的不同的概念：*authentication* 和 *authorization*。在开始本节之前，让我们花费一些时间来澄清这些概念。

定义认证（authentication）和授权（authorization）

认证和授权是安全模型里面作为系统访问控制的两个不同的概念。

认证的机制是提供一种辨别用户或者服务的方法，例如有时（并不总是）通过用户名和密码。在请求被接受之前，由用户（或者服务）输入的凭据和存储在数据库之中的凭据进行比较。

另外一方面，**授权**是执行访问控制规定的过程，它是一种定义认证用户（或者未认证的用户）在一系列物体上允许执行哪些动作的方法。

在 Mesos 上面，不仅可以开启对于用户和系统管理员的认证和授权，还可以对 framework 和 slaves 进行访问控制。本小节介绍组成 Mesos 集群安全和访问控制的几个功能：framework 的认证和授权、slaves 的认证和访问控制列表和 framework 的速率限制。

6.3.1 Slave 和 framework 的身份认证

在任何的生产系统中，能够有认证机器和应用程序的能力是至关重要的。在 Mesos 里面，认证能确保没有授权的 slaves 不会加入集群以及没有授权的 framework 不能消耗集群的资源。

幸运的是，Mesos 为 framework 和 slaves 提供了使能认证和配置认证的方法。它使用简单身份验证和安全层（SASL）framework，使用 CRAM-MD5 作为安全认证机制。

尽管我们大部分人都熟悉用户名和密码的认证，Mesos 在术语上可能有所不同；用户名成为 *principals*，密码被称为 *secrets*。值得注意的是 principal 是不同于 framework 用户（运行 framework 任务的 Linux 用户）和 framework 角色（用作资源保留）的。这个在本小节的后面提到。

运用 principal-secret 和 challenge-response 的认证概念，让我们看看怎样在 masters、slaves 和 framework 之间配置认证。对于你在此设置的 masters 的安全选项，你需要确保在多个 masters 上面是一样的。

Slave 的认证

Mesos masters 提供了几个允许 Mesos slaves 参与到集群的选项，包括对一组主机设置白名单或者还有需要 slaves 进行 principal 和 secret 的认证。这些配置选项在表 6.5 中列举出来了：

表 6.5 slave 认证的配置选项

配置选项	描 述
`whitelist`	用一个文件路径作为参数。文件里面是一些 masters 会通告资源的 slaves 清单（每行一个）
`credentials`	包含用户名和密码的磁盘上的一个文件路径（例如 file:///path/to/file）
`authenticate_slaves`	如果设置为 true，只有认证的 slave（在凭据文件里面有凭据存在）才被允许注册到 masters 里面。如果设置为 false，说明所有的 slaves 都可以注册进来

除了这些 master 范围的选项外，还需要提供凭证给 Mesos slave 到 masters 里面认证。--credential 这个选项配置一个磁盘上的文件，文件包含了 Mesos slave 的认证凭据。

我想最好是用一个简短的例子来展示一下怎么配置 slave 的认证。在 Mesos master 上面创建一个凭据文件包含下面内容：

```
{
  "credentials": [
    {
      "principal": "slaveuser",
      "secret":    "slavepass"
    }
  ]
}
```

你可以用空格分隔的用户名和密码创建凭证文件，每行一对：

```
$ echo -n "slaveuser slavepass" > /etc/mesos/secure/credentials
```

注意：几个热门的编辑器（Vim 和 Emacs）自动在文件的末尾插入一个新行，这个会导致认证的失败。你可以通过二进制模式打开文件明确地设置 *noeol* 选项以避免这种行为。另外，你可以使用 JSON 格式的凭证文件。

然后通过 –credentials 选项来配置 master 并重启服务：

```
--credentials=file:///etc/mesos/secure/credentials
```

提供给 slave 有认证的凭证，根据之前的格式创建一个凭证文件：

```
$ echo -n "slaveuser slavepass" > /etc/mesos/secure/slave-credentials
```

Mesos slave 上面的配置选项可能和 Mesos master 的稍微有点不同；在 slave 上面，你会用 --credential 这个配置选项来设置 slave 的凭证文件路径并重启服务：

```
--credential=file:///etc/mesos/secure/slave-credentials
```

通过参考这些指令，你就可以确保 slave 在加入集群之前一定要先进行认证。配置 framework 的认证也是采用相似的方法。

framework 的认证

正如你要求 slave 需要认证到 master 一样，你也可以使能 framework 的认证。为了使能 framework 注册到 Mesos 集群的认证，你需要在 masters 上面设置两个配置选项：

- --credentials——像之前说的，这个选项是一个包含用户密码的磁盘上

*的文件路径。无论你是认证 slaves 还是 framework，所有的凭据都是存储在*这个配置选项指向的文件当中的。

- `--authenticate`——将这个选项设置为 true，这样就只允许认证了的 framework 注册到集群当中。

重申一下我们上一小节提到的 slave 认证，创建一个凭证文件（或者添加到存在的凭证文件中）包含下面的内容：

```
{
  "credentials": [
    {
      "principal": "frameworkuser",
      "secret":    "frameworkpass"
    }
  ]
}
```

然后通过 --credentials 选项来配置 master 并重启服务：

```
--credentials=file:///etc/mesos/secure/credentials
```

提示：记得注意凭据文件的属主、属组和权限。由于 Mesos 当前的凭据是使用文本形式存储的，你需要确保只有拥有 root 级别权限的管理员才可以读取文件。

现在 framework 注册到 master 里面需要提供 principal 和 secret 才可以被认证。我们现在不需要担心配置单一的 framework，第 3 部分会包含更多的细节，会介绍 Marathon、Chronos 和 Aurora framework。

6.3.2 用户授权和访问控制列表

Mesos 会检查访问控制列表 (ACL) 来判断某个请求是否被允许执行。这些 ACL 由主谓宾三个维度组成：一个 *subject*（主语），及其所能在指定的一组 *objects*（宾语）上、执行的一个 *action*（谓语）。在之前认证小节提及的 framework 的 principal（或者用户），与你将在 ACLs 中使用的 principal 是相同的。请查看图 6.1：

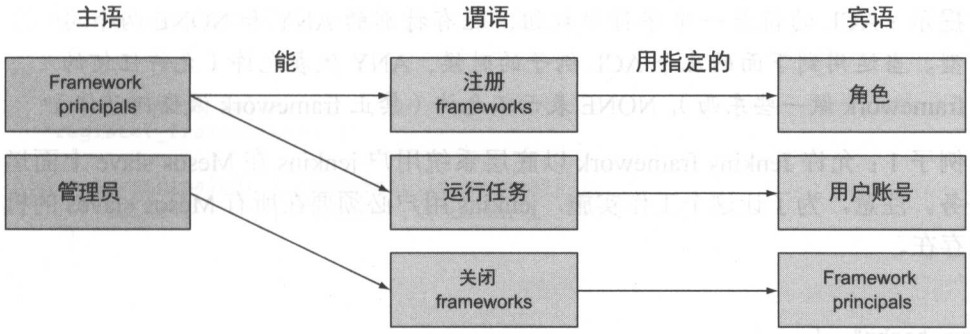


图 6.1 Mesos ACL 由可以在某些 objects 上面执行 action 的 subject 组成

为了清晰起见，我没有用 Mesos 词汇描述上图中的主谓宾，但希望你能够理解，每个部分都是为了执行请求而不可或缺的。依赖于 framework 的认证机制，这些 ACL 列表将允许或者阻止 framework 和用户在集群中执行某一类动作。在下面的几个小节中，我会提供几个怎样通过编写 ACL 来定义 framework 对集群访问权限的例子。

理解访问控制列表

与一些按照默认最小权限或者最高权限的系统相比，Mesos 的 ACLs 是根据定义的顺序一次匹配的：即只由第一条匹配这个请求的 ACL 来决定该请求是否被授权了。表格 6.6 提供了一个由主谓宾（subjects、actions 和 objects）组成的 ACLs 列表：

表 6.6 由 subjects、actions 和 objects 组成的 ACLs 列表

Subject	Action	Object
principals	register_frameworks	roles
	run_tasks	users
	shutdown_frameworks	framework_principals

作为后备，在 ACL 中也提供了一个叫做 permissive 的选项；这个选项定义了当请求没有匹配 ACL 里面的定义的时候要作出的默认行为。这个选项默认是 true，意味着动作是被允许的。

实施访问控制列表

访问控制列表在 Mesos masters 上面通过 --acls 这个选项配置。这个选项接收一个包含 ACLs 的文件路径（例如 file:///etc/mesos/secure/acl）或者 JSON 格式的 ACL。

我相信展示 Mesos 访问控制列表的最佳方法就是举例。下面的例子包括几个 ACL 的配置，还有它们要完成目标的简短介绍，展示了怎样限制访问 `run_tasks` 和 `register_frameworks` 这两个动作。

提示：ACL 的值是一串字符串数组，还有特别的 ANY 和 NONE 两个类型。当运用到下面的几个 ACL 例子的时候，ANY 代表允许（允许任何的 framework 做一些东西），NONE 表示不允许（禁止 framework 做任何动作）

例子 1：允许 Jenkins framework 以底层系统用户 jenkins 在 Mesos slave 上面运行任务。注意，为了让这个工作实施，jenkins 用户必须要在所有 Mesos slaves 的机器上存在。

```
{
  "run_tasks": [
    {
      "principals": { "values": ["jenkins"] },
      "users":      { "values": ["jenkins"] }
    }
  ]
}
```

例子 2：阻止 framework 作为 root 用户在 Mesos slaves 上面运行任务。

```
{
  "run_tasks": [
    {
      "principals": { "type":   "NONE"   },
      "users":      { "values": ["root"] }
    }
  ]
}
```

例子 3：为注册到某一个角色下面的 framework principals 创建一份白名单。在这个例子中，只有 marathon 才可以注册为 prod 角色；任何从其他 framework 过来注册到 prod 角色的请求都会被拒绝。

```
{
  "register_frameworks": [
    {
      "principals": { "values": ["marathon"] },
      "roles":      { "values": ["prod"]     }
    },
    {
      "principals": { "type":   "NONE"   },
      "roles":      { "values": ["prod"] }
    }
  ]
}
```

例子 4：为所有角色的全部 framework 创建一个全局的白名单。在这个例子中，仅有 chronos 这个 framework 以 batch 这个角色才可以注册进 masters 中。其他任何

的 framework 以任何角色注册都会是失败的。

```
{
  "permissive": false,
  "register_frameworks": [
    {
      "principals": { "values": ["chronos"] },
      "roles":      { "values": ["batch"]   }
    }
  ]
}
```

除了在 ACL 上授权特定的 framework principals 执行某一些例如以某一个身份注册到 Mesos master 里面的动作外，Mesos 也提供了另外一种限制从 framework 发送消息的速率限制办法。

6.3.3 framework 速率限制

在任何运行多个 framework 的部署环境当中，一个损坏的角色就可能将 master 淹没在消息当中，从而降低了高优先级的吞吐量，“生产水平”的 framework。Mesos 的 framework 速率限制通过 principal 限制某些特定 framework 的每秒查询次数和队列消息的数目，来保护这些高优先级的 framework。

通过限制一个给定的 framework 每秒处理的消息数目，你可以确保 Mesos 及时地响应其他的 framework。通过限制 master 队列里面的消息数目可以为 master 的内存提供某种保证。我会在本小节中更详细地说到这两个主意。现在查看图 6.2，你可以看到不同的速率限制和队列能力：

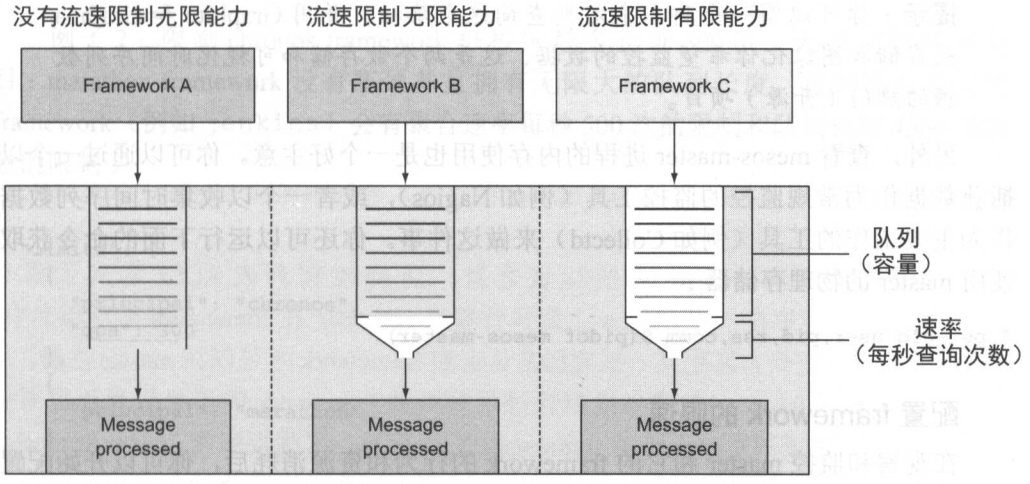


图 6.2 framework 速率和容量限制配置的组合

当考虑后面小节的监控和配置的时候，切记 framework 速率限制的目标并不是尽可能精确地模拟 framework 的行为，而是确保高优先级 framework 的吞吐量不会受低优先级的 framework 影响。

因为没有被配置速率限制的 framework 会尽量快地处理消息，对低优先级的 framework 进行优先速率限制是完全合理的，这让你知道这个功能是怎样在你环境中工作的。

监控 framework 和 master 的行为

我建议在进行速率限制之前先观察运行 framework 的特性。因此，进行配置 framework 速率限制的第一步就是监控为 framework principal 接收和处理的消息。每个 framework principal 的这些指标可以通过查询主 Mesos master 的 /metrics/snapshot API 路径获得。

要想知道怎样监控从每个 framework principal 接收和处理的消息，让我们看看这些 API 路径。会有一系列的跟随着 frameworks/<framework-principal> 这种格式的元素。在下面的 marathon 的例子当中，你可以看到 master 已经接收和处理了 254 条消息：

```
"frameworks/marathon/messages_processed": 254,
"frameworks/marathon/messages_received": 254,
```

当速率限制关闭（默认）的情况下，`messages_received` 和 `messages_processed` 返回的值应该是相等的（或者几乎相等），因为消息会在被接收后立马被处理。在你进行速率限制之前，你需要一直轮询查看这个 API 以得到你集群的一个准确活动情形。

> **提示**：你可以写一个脚本来定期查询这个路径，并用 Graphite 和 Grafana 来存储和图像化你希望监控的数据，这是两个做存储和可视化时间序列数据的热门（开源）项目。

另外，查看 mesos-master 进程的内存使用也是一个好主意。你可以通过一个以捕获数据作为常规监控的监控工具（例如 Nagios），或者一个以收集时间序列数据作为主要工作的工具（例如 Collectd）来做这件事。你还可以运行下面的命令获取使用 master 的物理存储器：

```
$ ps -efo user,pid,rss,comm $(pidof mesos-master)
```

配置 framework 的限速

在观察和监控 master 和它的 framework 的行为和资源消耗后，你可以开始配置 framework 的速率限制了。但是，让我们先定义一些 Mesos 用来做速率限制的术语：

- `principal`——framework 的标识（注意：多个 framework 可以有同一个标

识）。

- qps——限制的速率，也就是每秒的查询数。如果一个 principal 的这个配置忽略的话就是对这个 principal 没有限制。
- capacity——master 队列里面能存放的消息数量；被接收了但是没被处理的消息。只有在 qps 启用的时候才会生效。
- aggregate_default_qps——没有配置速率限制的 framework 的总共速率值。
- aggregate_default_capacity——没有配置速率限制的 framework 的队列里面总共的消息数目。只有在 aggregate_default_qps 启用的时候才生效。

就好像在 ACLs 那个小节一样，我想最好展示速率限制的方法就是举两个例子。下面的每一个配置都代表了 master 配置选项 --rate_limits 的有效值。

例子 1：限制 jenkins framework 最多是每秒查询 100 次，队列里面最多是 6000 条消息。其他的 framework 没有限制，拥有无限大的队列数目：

```
{
  "limits": [
    {
      "principal": "jenkins",

      "qps": 100,
      "capacity": 6000
    }
  ]
}
```

例子 2：限制 chronos framework 每秒钟最多查询 300 次，无限大的队列数目；marathon framework 没有瓶颈并且拥有无限大的队列长度。任何没有定义的 framework（例如 jenkins）会有聚合速率每秒 500 次的限制和队列长度 4500 条消息的限制：

```
{
  "limits": [
    {
      "principal": "chronos",
      "qps": 300
    },
    {
      "principal": "marathon"
    }
  ],
  "aggregate_default_qps": 500,
  "aggregate_default_capacity": 4500
}
```

如果 framework 超过其配置的速率限制，并随后超过其配置的容量，则有一个错误消息被发送回该 framework。framework 开发人员可以使用此事件来触发他们 framework 调度程序中的行为或行动来处理这种情况。

> **提示：**未来的 Mesos 发行版本能够在 master 的消息队列满了的时候通知 framework。这样的话 framework 就能根据消息开始被队列阻塞的时间来作出反应了，而不是等到超过队列限制才作出反应。可以通过查询这个网址 https://issues.apache.org/jira/browse/MESOS-1664 获得这个功能的开发进展。

正如你在前面的几个章节中看到的，结合 framework、身份验证和速率限制为你提供了许多对连接到集群 framework 的控制。

6.4 小结

在本章，你学到了怎样在生产环境中监控和管理 Mesos 集群。你学到的主题包括监控 Mesos 和 ZooKeeper，增加和移除 Mesos master，还有实施访问控制列表。如下一些事情需要记住：

- Mesos masters 和 slaves 都提供了一个基于 JSON 格式的 REST API，其包含关于集群或者单个节点有价值的信息。一些有用的路径列表可以在下面找到：http://mesos-master.example.com:5050/help 和 http://mesosslave.example.com:5051/help。
- 为了确保你的监控请求能转发到主 Mesos master，你可以考虑由 master 提供的重定向路径接口 /master/redirect。另外，你也可以在你的监控系统和 Mesos masters 之间放置一个 HAProxy 的实例。
- 为了监控和管理 ZooKeeper 集群，Netflix 团队创建了一个名叫 Exhibitor 的开源工具。ZooKeeper 也可以被一组四字命令监控，例如 ruok 和 mntr。项目的维护者们也在项目的 src/contrib/ 目录下面列入了很多监控脚本。
- 在一个现有集群增加 Mesos master 的时候，要确保在新 master 上线之前配置好集群的法定数目。
- 除了 slave 和 framework 的认证之外，访问控制列表 ACLs 定义了哪个 *subjects* 能够在一组 *objects* 上面执行哪些 *actions*。在 framework 的环境当中，ACLs 确定只有匹配某一系列标准认证的 framework 才可以在 master 注册。
- Mesos 的 framework 速率限制通过限制其他 framework 的消息处理数目来保护高优先级和生产水平的 framework 正常运行。这些指标在 /metrics/snapshot 这个路径上面展现出来，并且通过它们的 *principal* 来区分。

这些总结了本书的第 2 部分。第 3 部分讲述了一些热门的开源 framework，这

些 framework 使得你能够在 Mesos 集群上面运行应用程序和调度任务，还介绍了 Mesos 的 API，也提供了开发你自己的 Mesos framework 的指引。

在第 7 章中，你可以学习到热门的 Marathon framework，其用于在 Mesos 集群上运行应用程序和 Docker 容器。

第3部分

运行Mesos

现在你已经掌握了在生产环境部署 Mesos 的方法，而本书的第 3 部分首先涵盖了如何使用热门（和开源的）的 Mesos framework 来部署应用与调度作业。接着，你将学习到集群内的服务发现，以及用户访问通道的负载均衡。最后，我会介绍 Mesos 的 APIs 和一些部署自制 Mesos framework 的例子。

7 使用Marathon部署应用

本章内容

- 安装和配置 Marathon
- 部署应用和 Docker 镜像
- 使用 HAProxy 实现服务发现和服务路由

在本书的第 1 和第 2 部分，你学习了 Aapche Mesos 项目，并知道如何配置用于生产的 Mesos 集群。而本章作为第 3 部分的开端，在此你将开始通过部署应用和调度任务让你的 Mesos 集群投入工作。

本章将为你介绍 Marathon 这个由 Mesosphere 公司开发的，热门且开源的 Mesos framework。通过 Marathon，我们可以部署长期运行的服务和应用（包括 Docker 容器）。本章会通过实际的例子使你逐步熟悉 Marathon 和应用管理。

7.1 了解Marathon

到目前为止，你已经通过运行多种不同的 framework 来探索 Mesos，并且知道在不需要静态分区的数据中心中，如何更高效地管理数据中心的资源。你也了解到特定的 Mesos 应用，例如 Jenkins 和 Spark，可以直接连到 Mesos 集群上并且运行任

务。但对于更传统的应用，或者通过 Docker 镜像部署的应用，又如何实现呢？

如果你将 Mesos 类比为一个操作系统的内核，那么 Marathon 相当于服务管理系统。在 Linux 操作系统里面，通常代表 *init* 系统。Marathon 通过 Linux cgroups 和 Docker 容器来部署应用作为长期运行的 Mesos 任务。或许更准确地说，它也是一个私有的用于部署应用的“平台即服务（PaaS）”。Marathon 是通过启动一定数量的长时间存活的 Mesos 任务作为应用实例来实现的，如图 7.1 所示。

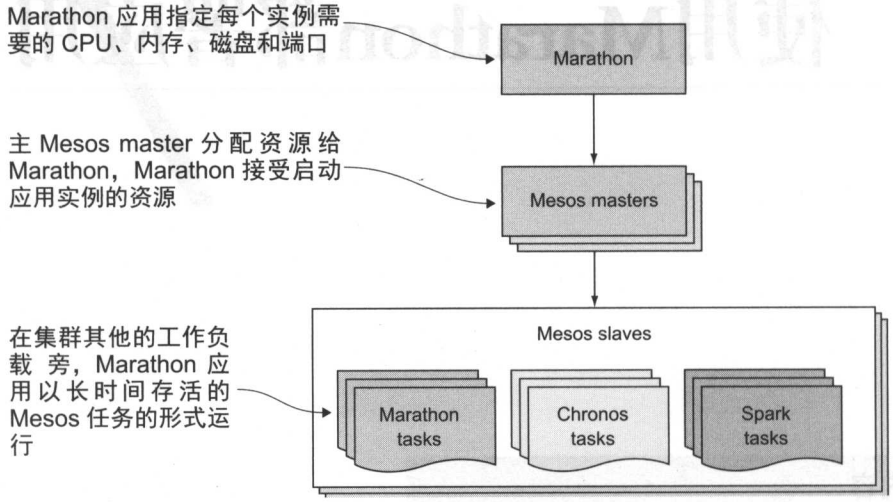


图 7.1 Marathon 启动应用实例作为长时间存活的 Mesos 任务

Marathon 允许你指定一个应用中每个实例所需要的资源，以及你想要运行的实例数量。和 systemd 与 Upstart 这些现代服务管理器类似，Marathon 会自动使用集群中可用的资源来重启失败的任务。如果一个 Mesos slave 挂了，或者你应用中的一个实例崩溃或者退出了，Marathon 也会自动启动一个新的来代替失败的实例。Marathon 也允许用户在部署过程中指定该应用与其他服务的依赖关系，所以你可以确保一个应用实例在启动之前，所依赖的数据中心实例需要先启动，并且通过了健康检查。

注意：本章讲的 Marathon 版本为 0.10.1。

Marathon 包含着大量的特性来满足大多数应用管理的场景，下面几点是最值得一提的。

- 使用依赖和健康检查来管理应用和应用组。
- 使用指定的容量要求来回滚应用升级。
- 强大的 Web 接口和 REST 风格的 API。
- 高可用（用 ZooKeeper 来进行 leader 选举和协调）。

下面的小节会介绍最热门的特性，并且展示 Marathon 是如何管理应用和部署的。你将学习到如何将 Marathon 和例如 Haproxy、Mesos-DNS 等工具组合起来处理服务发现和服务路由。

7.1.1 探索 Marathon 的 Web 接口和 API

为了更熟悉 Marathon，我们先讨论和它交互的两种主要方式：Web 接口和 REST API。

探索 Web 接口

Marathon 有直观的 Web 接口，用来管理应用和查看应用部署状态。让我们快速地看一下图 7.2Wed 接口的 Apps 页面。

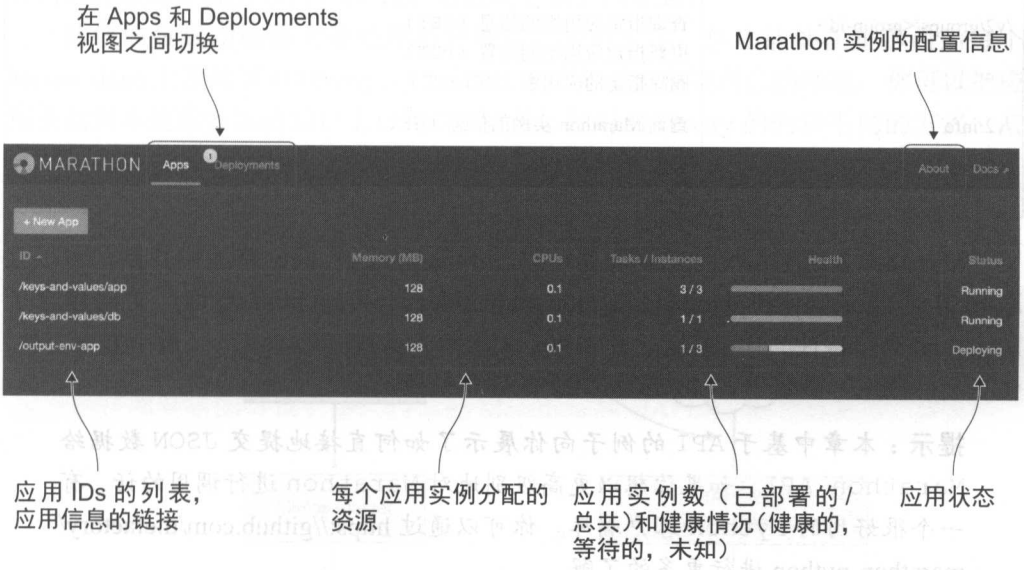


图 7.2 Marathon Web 接口总览

尽管 Web 接口有助于观察和管理应用和部署，但还是存在一些局限的。例如你不能通过 Web 接口来启动 Docker 容器，或者不能使用更高级的特性，如最小健康容量或者健康检查的设置。对于这些特性，你需要使用 Marthon 的 API 来满足管理应用。

探索 REST API

考虑到大量的企业组织都利用持续集成（CI）和持续发现（CD）来实现他们的应用自动化部署，Marathon 提供了丰富的基于 JSON 的 REST API 来帮助他们更轻松地部署新应用或者已有应用的新版本。表 7.1 包含了重要的 Marathon API 路径

和操作它们的 HTTP 方法。本章的剩余部分，你将会使用到这些列出的 API 路径。因为基于使用的 HTTP 方法，每个路径的调用结果是不同的，所以表格包含了每个 API 路径的多种行为，取决于括号内的 HTTP 方法。

表 7.1 重要的 Marathon API 路径

API 路径	描 述
/v2/apps	查询 Marathon 实例上面所有的应用（GET） 创建新的应用（POST）
/v2/apps/<app-id>	查询指定 APP 的信息（GET） 更新指定 APP 的配置（PUT） 删除指定 APP（DELETE）
/v2/groups	查询 Marathon 实例上面所有的应用组（GET） 创建新的应用组（POST）
/v2/groups/<group-id>	查询指定应用组的信息（GET） 更新指定应用组的配置（PUT） 删除指定的应用组（DELETE）
/v2/info	查询 Marathon 实例的信息（GET）
/v2/leader	查询当前 Marathon 集群 leader 的主机名、端口（GET） 造成当前的 leader 退位，并触发一次新的 leader 选举（DELETE）

Marathon REST API 的完整文档可以在 Marathon 的 /help 路径中找到，可以互联网访问 https://mesosphere.github.io/marathon/docs/rest-api.html 查询。文档包含了 Marathon 高级特性的详细信息，例如 minimumHealthCapacity（最小健康容量，供更新回滚用）和基于 HTTP 或者 TCP 的健康检查。

> **提示：**本章中基于 API 的例子向你展示了如何直接地提交 JSON 数据给 Marathon API。如果你想以更高级别地对 Marathon 进行调用的话，有一个很好用的 Python 客户端库。你可以通过 https://github.com/thefactory/marathon-python 进行更多的了解。

稍后我会列举一些实践的应用管理场景，但现在，让我们来关注由现代应用架构组成的大量服务，它们是如何做到互相通信的。在本书中，我称之为服务发现和路由。

7.1.2 服务发现和路由

现代应用通常是由多个服务或者多个层级组成的。最普遍的现象是，由一个负载均衡器来负责转发从用户的请求到几个 Web 服务器。前端的 Web 应用通常消耗 RESTful 的 API，并且运行在负载均衡器后端服务器上面。相应地，这些 API 服务器就当做一个或者多个后端数据库的抽象。

无需赘言，使这些服务之间能够互相通信是很关键的。使用一个指定的主机名来关联服务在传统环境下相对容易，但是在 Mesos 环境下，是相对复杂的。因为只要计算资源充足，容器能够运行在集群中的任何地方。为了更新负载均衡器的配置，你需要知道某个特定应用的所有实例所处的位置，而这一点是很难的。

在 Mesos 和 Marathon 的生态圈，我们把这个问题（和解决方法）叫做服务发现。幸运的是，目前有多个可用的工具来处理服务发现和服务路由。本节将涵盖两种最热门的选择：HAProxy 和 Mesos-DNS。

使用 HAProxy 做服务路由

如在本章前面提到的，Marathon 可以通过其 REST API 向外提供在应用上面运行实例的信息。通过一些小工具，你可以利用这个信息动态地建立 HAProxy（一个热门、轻量级开源的 HTTP 和 TCP 负载均衡器）的配置。

传统的负载均衡器主要处理用户流量，相对来说，你还能够在集群内的每个 Mesos slave 上面部署 HAProxy。然后取决于哪些服务需要通信的情况，你可以把应用关联到本地服务器的端口上。图 7.3 演示了使用 HAProxy 的两种不同的方式：其一用于内部集群通信，其二用于处理来自于用户的向内请求连接。

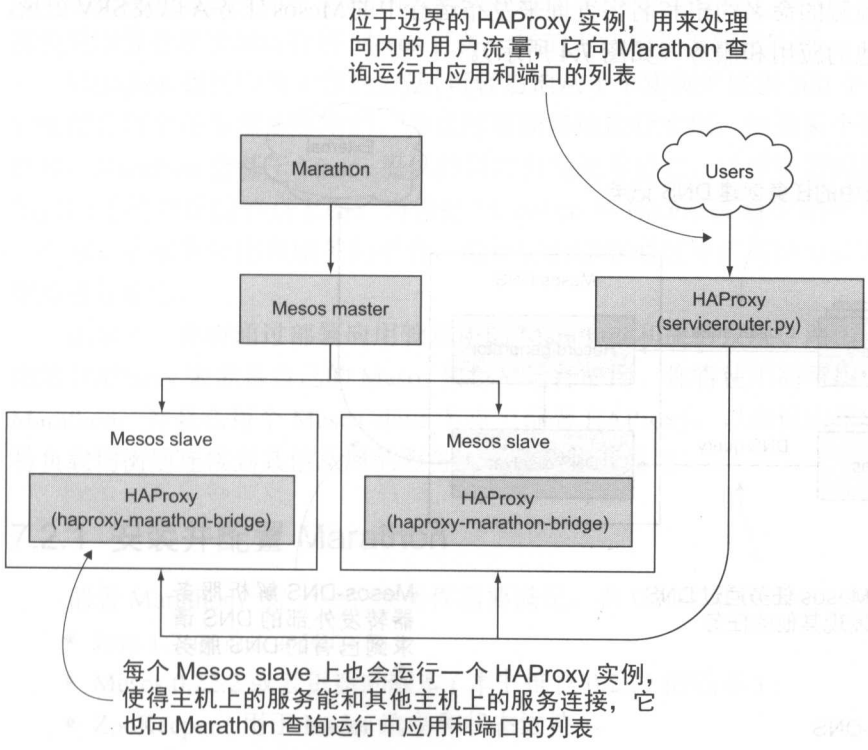


图 7.3 使用 HAProxy 做服务路由

也许是因为 HAProxy 的热门和稳定，Marathon 发布了两个脚本来周期性地（通过 Cron）查询 Marathon API 获取到运行中的应用实例信息。通过利用这个信息来构建 HAProxy 的配置并且重载服务。两个脚本如下：

- *haproxy-marathon-bridge*——一个短小的 shell 脚本，基于 Marathon 的应用信息来构建 HAProxy 的配置。每当配置有变化的时候，它会自动重载 HAProxy 的服务。
- *servicerouter.py*——类似于上述脚本，但在生成的 HAProxy 配置之上允许有更多的控制。

每个脚本都可以从 Marahotn 的项目版本库上下载，地址是 https://github.com/mesosphere/marathon/tree/v0.10.1/bin 。两个脚本之间有细微的差别，取决于你需要多大的控制或者生成的 HAProxy 配置的需求。现在还不用担心下载它们，因为在接下来的章节里，你将学习如何部署 HAProxy 来实现服务路由。

使用 Mesos-DNS 做服务发现

Mesos-DNS 是由 Mesosphere 开发的一个开源项目，它是专为 Mesos 集群设计的无状态域名解析（DNS）服务。和通常的 DNS 服务器类似，它允许应用和服务通过使用可预测的命名约定和名字查询来发布运行中的Mesos任务A以及SRV记录，最终找到其他的应用和服务（如图 7.4 所示）。

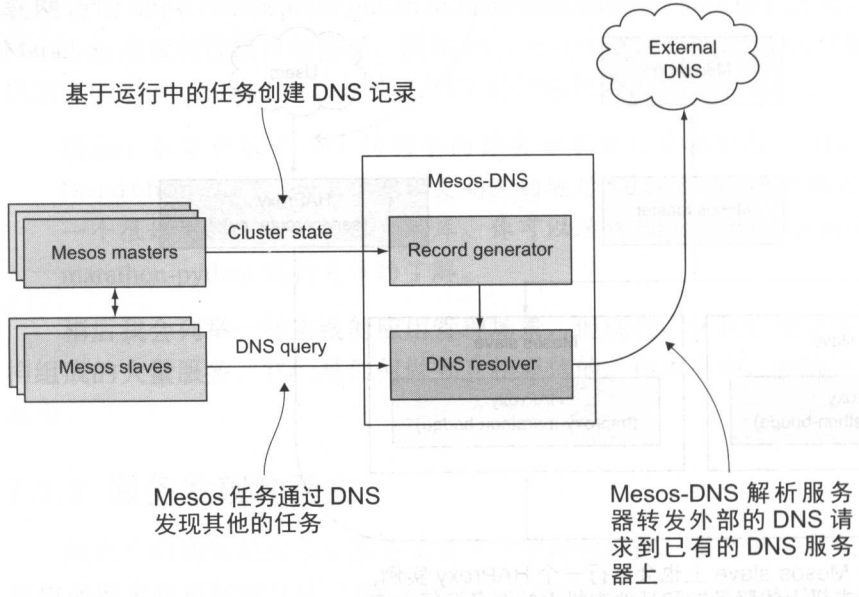


图 7.4 Mesos-DNS

和前面提到的 HAProxy 不同，Mesos-DNS 直接使用 Mesos 进行工作，不依赖于 Marathon。这使得使用不同 Mesos framework 工作的服务之间能够互相通信。

提示：图 7.4 展示了使用 Mesos-DNS 作为解析器，可以转发外部（未知）的 DNS 查询请求。如果你原本已经拥有一套 DNS 架构，那么也可以做相反的事情：在原有的 DNS 服务器上面配置外部的 DNS 查询（默认为 .mesos 域）转发到 Mesos-DNS 上面。

尽管本节既涵盖了 HAProxy，又涵盖了 Mesos-DNS，但由于 DNS 是大部分人都熟悉的概念，所以我将在本章接下来的内容中演示使用 HAProxy 进行服务路由。如果你对 Mesos-DNS 项目的更多细节感兴趣，请前往项目页面：http://mesosphere.github.io/mesos-dns。

7.2 部署Marathon和HAProxy

Mesos 集群通过容器内独立的任务承载所有的工作。在容器内运行应用减少的消耗意味着你能更高效地使用数据中心资源。但你需要采用管理由长期运行的 Mesos 任务组成的应用实例的方法来部署应用和服务。但为了部署应用和服务，你需要采用像管理长期运行的 Mesos 任务一样的方法来管理应用实例。

Marathon 提供应用扩容的方法，可轻易地从 1 个实例扩展到 100 个（甚至更多），它能保证每个任务都是启动的，并且尽可能多地运行实例。如果某个任务或者节点挂掉，Marathon 会基于 Mesos 提供的资源自动地重启它，这意味着应用实例能够按需自动迁移到新的节点上面。当你把 Marathon 和 HAProxy 组合起来后，你会得到一个强大的部署应用和服务的平台，确保它们能够通过可扩展的方式和其他从属的服务进行通信。

在本节，你将通过部署应用管理用的 Marathon 和服务发现、路由以及负载均衡用的 HAProxy 来准备自己的 Mesos 集群来运行应用。你将使用高可用的方式来部署 Marathon，并且在每个 Mesos slave 节点上部署 HAProxy，以确保应用实例能够很容易负载均衡地连接到其他从属的服务上。让我们开始吧。

7.2.1 安装并配置 Marathon

部署 Marathon，有一些先决条件需要满足。在 0.10.1 的版本中，包含如下条件：

- Java 1.7 或者更新的版本；
- Mesos 0.22.2 或者更新的版本（本书推荐 0.22.2 的版本）；
- ZooKeeper（用于 leader 选举和状态维持）。

为了让 Marathon 实现高可用，你需要像第 3 章中部署奇数个 Mesos master 一样，在多个节点上部署 Marathon。在大部分场景下，部署 3 个 Marathon 实例是合适的，并且每个实例都会使用 ZooKeeper 集群来实现 leader 选举和状态维持。

为了更简便地安装，你可以考虑把 Marathon 部署在 Mesos master 相同的机器上面。为了让本节的例子更容易理解，我们假设在第 3 章中你已经部署好了 Mesos master，你将把 Marathon 安装在了这上面。

Marathon 独立的 DNS 域名

在一个高可用的 Marathon 部署中，多个实例被部署在多个主机上。基于管理和应用自动化部署的意图，你会考虑创建一个独立的 DNS 记录指向到你的 Marathon 集群。幸运的是，Marathon 有这么一个特性：无论请求从哪里来，一旦被发送到非 leader 节点，该请求都会被透明地代理到 leader 节点进行处理。

你可以使用两种方法通过单独的域名来连接 Marathon 集群。

- *DNS* **负载均衡**——通过多个指向每个 Marathon 实例的 A 或者 CNAME 记录创建独立的 DNS 域名。
- *HTTP* **负载均衡**——使用负载均衡器（例如 HAProxy）来转发 Marathon 的连接，创建一个指向该负载均衡的 DNS 域名。

每个选项都有优点和缺点。举例来说，DNS 负载均衡便于配置，但如果 Marathon 其中的一个实例不可用，可能会造成失败的连接尝试。另外，添加一个 HAProxy 负载均衡器，可以自动地移除后端池中失效的实例，但需要花费人力在客户端和 Marathon 之间添加该服务。

接下来，让我们继续安装和配置 Marathon 吧。

安装 Marathon

安装 Marathon 最简单的方法是使用第 3 章中安装 Mesos 期间配置的 Mesosphere 安装包仓库。根据你的 Linux 发行版本，运行下面的命令来安装 Marathon：

- RHEL 和 CentOS: `sudo yum install marathon-0.10.1-1.0.416.el7`
- Ubuntu: `sudo apt-get install marathon= 0.10.1-1.0.416.ubuntu1404`

尽管可以从源码下载和安装 Marathon，但这个过程比使用安装包管理更复杂。所以，本书不会涉及这些构建指引。但如果你有兴趣的话，可以查看项目的“Getting Started”文档 (http://mesosphere.github.io/marathon/docs) 来获取最新的指引。

配置 Marathon

和 Mesos 类似，Marathon 也有一些配置规范，这取决于你的部署方式和偏好。因为 Mesosphere 开发并打包了 Marathon，所以其基于文件的配置（类似于 Mesosphere 的 Mesos 安装包）可能是上手 Marathon 最简单和直截了当的方式，你可通过在 /etc/marathon/conf/ 目录下创建文本文件来配置 Marathon。

Marathon 有多种配置选项，如 https://mesosphere.github.io/marathon/docs/command-line-flags.html 所指定的。表 7.2 是在你部署 Marathon 时，我找出来的最值得一提的具体配置选项。

表 7.2 重要的 Marathon 配置选项

配置选项	描　述
`master`	Mesos master 所使用的 ZooKeeper URL。如果 /etc/mesos/zk 是存在的 (Marathon 部署在 Mesos master 上)，那么该文件的值会被自动使用
`zk`	Marathon 用于 leader 选举和状态的 ZooKeeper URL。例如：zk://host1:2181/marathon 。如果 /etc/mesos/zk 是存在的 (Marathon 部署在 Mesos master 上)，Marathon 会使用和 Mesos 一样的 ZK 主机和端口，但是会创建自己的 / marathon znode 节点
`hostname`	指定 Marathon 实例的 DNS 域名或者 IP 地址
`mesos_role`	如果设置，以指定的角色将 Marathon 注册到 Mesos 集群中（详见第 6 章）
`mesos_authentication_principal`	framework 认证时的 Mesos 用户（详见第 6 章）
`mesos_authentication_secret_file`	framework 认证时包含密钥的文件路径（详见第 6 章）

Marathon 也支持 SSL 和基础认证，我特意把这个特性从表 7.2 中抽离，是因为有些用户可能更喜欢在多个 Marathon 实例前端的负载均衡器设置认证。如果你对启用 SSL 和 Marathon 自身实例的认证感兴趣的话，请查看 https://mesosphere.github.io/marathon/docs/ssl-basic-access-authentication.html。

提示：某些邮件列表上面的用户遇到过如下的问题，Mesos 调度驱动器绑定在错误的网络接口上，这会造成 Mesos master 和 framework 之间的连接问题。默认情况下，Mesos 本地库会绑定映射到操作系统 FQDN 域名（等同于 hostname -f）的 IP 地址。为了确保调度驱动器绑定到正确的接口上，需要保证主机的 DNS 记录（或者 /etc/hosts 的条目）是被正确配置的，或者可以通过手动设置 $LIBPROCESS_IP 的环境变量来指定 IP 地址，该地址位于你希望 Mesos master 通信使用的网络接口上。

在每个Mesos master安装和配置Marathon后，你可以通过运行sudo service marathon start命令来启动服务。在服务启动完成后，你可以通过http://mesos-master.example.com:8080访问Marathon的Web接口。

7.2.2 安装并配置HAProxy

以我们前面所覆盖的服务发现和路由内容为基础，你将通过在集群中的每一个Mesos slave上安装和配置HAProxy来处理各种不同的应用和服务之间的网络流量。使用Marathon分布部署的haproxy-marathon-bridge脚本会基于Marathon上可用的信息动态地生成HAProxy的配置文件。这将允许Marathon应用连接到本地主机的一个端口上，同时也自动地接入了一个依赖服务的运行实例。

此外，你还需要考虑如何处理用户端进入的流量。为此，我假设该运行servicerouter.py的HAProxy实例成为一个专门处理用户端进入流量的单独的机器。这让我能够用最好的方式来讲解如何部署负载均衡器的细节。

安装HAProxy

为了维持Marathon项目提供的脚本的兼容性，你将会在每个Mesos slave上安装1.5.x最新版本的HAProxy。最简单的方法是使用操作系统的包管理器。

注意：在Ubuntu 14.04上安装HAProxy 1.5.x，首先你需要通过取消/etc/apt/sources.list相关的注释符来启用trusty-backports。在这之后，请确保你运行sudo apt-get update命令来更新安装包的列表。

在Enterprise Linux或Ubuntu上，运行下面的命令来安装HAProxy：

- 在RHEL和CentOS上——sudo yum install haproxy；
- 在Ubuntu上——sudo apt-get -t trusty-backports install haproxy。

使用haproxy-marathon-bridge来动态配置HAProxy

为了促进集群内部服务间的通信，你将使用Marathon提供的haproxy-marathon-bridge脚本动态地创建HAProxy的配置，以及当变化发生时，自动地重载HAProxy服务。通过在每个节点上安装HAProxy，并且自动和动态地重配服务，一个应用被允许通过连接到本地服务器上应用的servicePort这种可伸缩的方法与另外的应用进行通信。为了让这个概念更直观，请看一下图7.5。

安装特定操作系统的haproxy-marathon-bridge，请从Marathon的GitHub库上面下载该脚本，并且使用内置的install_haproxy_system函数：

```
$ curl -LO https://raw.githubusercontent.com/mesosphere/marathon/
➥ v0.10.1/bin/haproxy-marathon-bridge
$ chmod +x haproxy-marathon-bridge
$ ./haproxy-marathon-bridge install_haproxy_system <host1> [host2] [...]
```

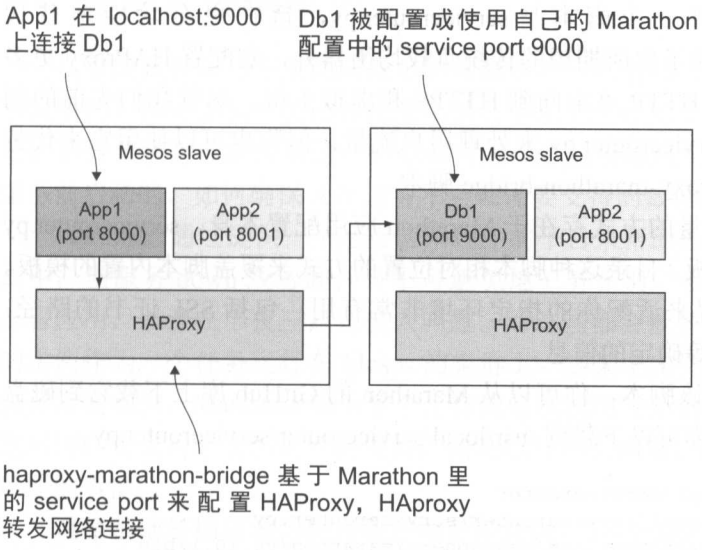


图 7.5 部署 HAProxy 来路由内部集群网络的流量

注意：对基于 Debian 的系统，haproxy-marathon-bridge 会默认系统中已经安装了 aptitude 工具，如果实际上没有安装的话，你可以通过命令 `sudo apt-get install aptitude` 来进行安装。

一路执行到这，你已经完成了如下操作：

- 把脚本安装到 /usr/local/bin/haproxy-marathon-bridge；
- 添加 `host1`，`host2` 和其他的 Marathon 主机到位于 /etc/haproxy-marathon-bridge/marathons 的配置文件上；
- 在 /etc/cron.d/haproxy-marathon-bridge 上创建了一个 Cron 任务，每分钟执行一次该脚本；
- 创建或者修改了位于 /etc/haproxy/haproxy.cfg 的配置文件，并重启了 HAProxy 的服务。

Haproxy-marathon-bridge 脚本添加应用的实例到 HAProxy 的配置文件，如果当前没有任务运行，则该文件为空。否则，HAProxy 会被用于应用间进行负载均衡连接。

由于 Cron 脚本是每分钟执行一次，如果在这一分钟的窗口里面，有某个 Mesos slave 下线或者某个任务（应用程序的一个实例）失败了，HAProxy 的健康检查会自

动地检测出异常并且停止把流量发给该实例。在下一个 Cron 任务执行时，失败的实例将会从 HAProxy 的配置中被移除。

使用 servicerouter.py 动态配置 HAProxy

Marathon 包含的另外一个脚本是 servicerouter.py，旨在更全功能地替代 haproxy-marathon-bridg。除了你所期望的传统负载均衡器外，它配置 HAProxy 更多的特征。例如 SSL 终端，HTTP 重定向到 HTTPS 和虚拟主机。尽管我们先前的例子在边缘节点上使用了 servicerouter.py 来处理用户流量，但你也可以使用它来代替每个 Mesos slave 上的 haproxy-marathon-bridge 脚本。

脚本的配置以环境变量的方式存在于 Marathon 应用配置本身。servicerouter.py 也允许管理员通过使用模板 / 目录这种脚本相对位置的方式来覆盖脚本内置的模板。这对于修改 HAProxy 配置来适配你的指定环境非常有用，包括 SSL 证书的路径，或者首选的负载均衡策略等确定的信息。

在特定的系统上使用该脚本，你可以从 Marathon 的 GitHub 库上下载它到磁盘上面常见的路径。例如，你可以下载到 /usr/local/servicerouter/servicerouter.py。

```
$ sudo mkdir -p /usr/local/servicerouter
$ sudo curl -L -o /usr/local/servicerouter/servicerouter.py
➥ https://raw.githubusercontent.com/mesosphere/marathon/v0.10.1/bin
➥ /servicerouter.py
$ sudo chmod +x /usr/local/servicerouter/servicerouter.py
```

你可以通过输入某些命令行参数来使用这个脚本，如指定不同 Marathon 实例的 URL 和保存 haproxy.cfg 文件的路径：

```
$ ./servicerouter.py --marathon http://marathon.example.com:8080
➥ --haproxy-config /etc/haproxy/haproxy.cfg
```

请执行如下命令来获知完整的使用方法：

```
$ ./servicerouter.py --longhelp
```

在你拥有一个用来构建满足个人喜好的 HAProxy 配置的脚本后，最好再创建一个本地的 Cron 任务来确保这个脚本在可预测的时间表内执行。虽然 HAProxy 会自动地把不健康的实例从负载均衡池中移除，但是确保配置信息总是从 Marathon API 获取的最新数据这点也是很重要的。

让我们在 /etc/cron.d/servicerouter 创建一个类似如下的 Cron 任务吧：

```
* * * * * root /usr/local/servicerouter/servicerouter.py <args>
```

现在你已经完成了安装和配置 Marathon 和 HAProxy，同时也包括了 haproxy-

marathon-bridge 和 servicerouter.py 两个脚本。让我们深入地使用 Marathon 来创建和部署你的第一个应用吧。

7.3 创建并伸缩应用

在系统操作员与开发团队众多的问题中，应用管理是其中一个。你是如何部署新应用的？你是如何升级已存在应用的？你是如何根据需求的增多与减少，来轻松地扩容或者缩容你的应用的？你是如何确保你的前端应用不会在所需要使用的 API 服务前启动的，如何确保 API 服务不会在需要支持的数据库前启动的？

Marathon 通过提供一个运行应用程序的平台来帮助运维和开发团队。在传统方式中，当服务 down 或者硬件（不可避免地）异常时，我们需要补充应用服务器，部署应用，以及在半夜三点还被人叫醒。而这些都会被替代，Marathon 把每个应用的实例作为一个任务运行在 Mesos 的集群上。当某个实例失败的时候，它会自动重启异常的实例。Marathon 确保整个工作流是从应用开始的，而非服务器。只要你创建应用，定义每个实例需要的资源，指定运行的实例数，Marathon 和 Mesos 会负责后面的事情。

遵循基础架构应该为应用程序服务的观念，Marathon 是通过使用 Mesos 内置支持的容器来轻松部署你的应用的。让我们开始聚焦使用 Linux 和 Docker 容器来部署 Marathon 应用，需要怎么做吧。

7.3.1 部署简单的应用

在 Marathon 的术语中，应用部署被定义为一组要达成的行动，如下：

- 开始（创建）或者停止（销毁）一个或者多个应用。
- 伸缩应用实例（n>1），或者在保留应用配置的情况下完全挂起（n=0）。
- 更新（修改一个或者多个应用的配置）和回滚整个基础架构的所有配置变化。

在写这一章时，我觉得最好要提供一些部署应用的现实场景，本书的补充材料包含了一个叫做 OutputEnv 的示例应用。这个简单的 Ruby 网页应用在 Web 页面上面输出指定的应用实例（Mesos 任务）的环境变量。

部署 OutputEnv 示例应用

因为 OutputEnv 是一个简单的应用（它不在 Docker 中运行，也不依赖于外界的服务、健康检查以及主机约束），所以让我们通过 Marathon 的 Web 接口来部署这个应用吧。

注意：因为 OutputEnv 不在 Docker 中运行，所以你必须在每个 Mesos slave

上安装有 Ruby (1.9.3 或者更高) 和 Bundler 来部署应用。Rudy 可以通过系统包管理器来安装，Bundler 可以通过执行命令 `sudo gem install bundler` 来安装。

从先前章节中提到 Apps 的主要页面上，单击 New App 按钮就会弹出如图 7.6 的对话框。

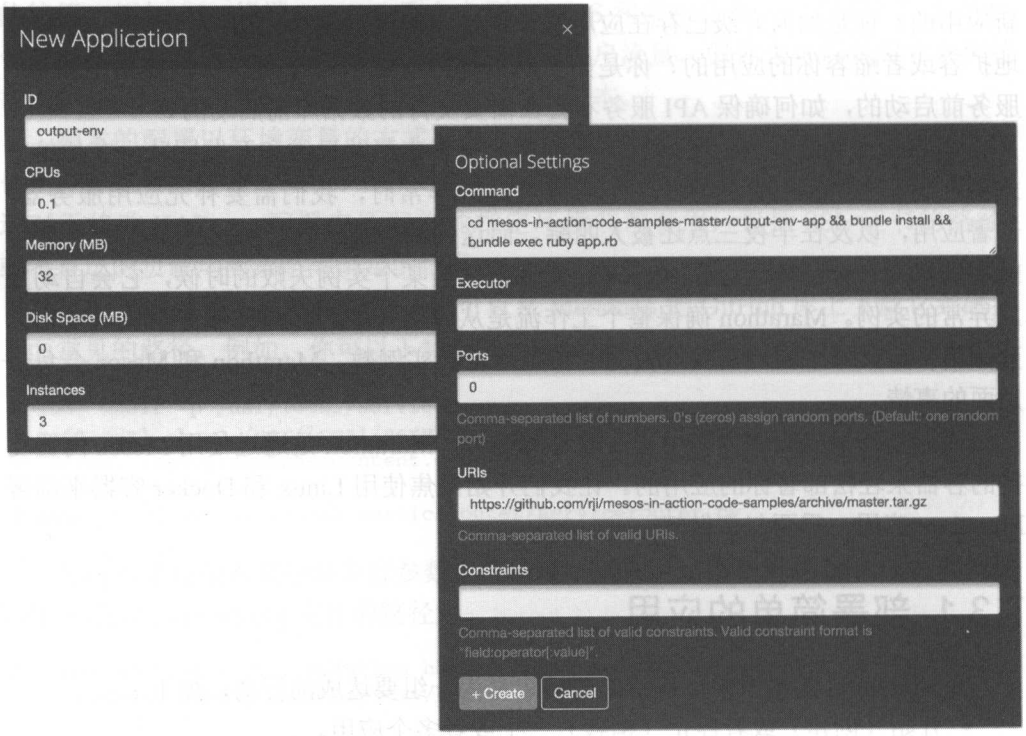


图 7.6 在 Marathon 网页接口上创建 OutputEnv 应用

在这里，你可以给应用命名，为每个实例设置要求的资源，并且设置实例的数量。在可选配置部分，你可以提供下载应用程序的 URI 和你使用的去运行它的命令。通过图 7.6 输入的配置，你可以从本书的 GitHub 版本库上面下载 OutpuEnv 应用，并且在你的 Mesos 集群中运行它。如果你在 URIs 区域中定义的是一个压缩文件（例如 zip 或者 tar.gz），Mesos 提取器在沙盒中会自动为你解压该文件。你可以单击 Create 按钮来开始部署应用。

管理 OutputEnv 实例应用

在创建应用之后，你会重返 Apps 的主页面。单击 Output-Env 应用会把你带到应用管理页面，在这个页面里，该应用所有的 Mesos 任务都会被列出，如图 7.7 所示。

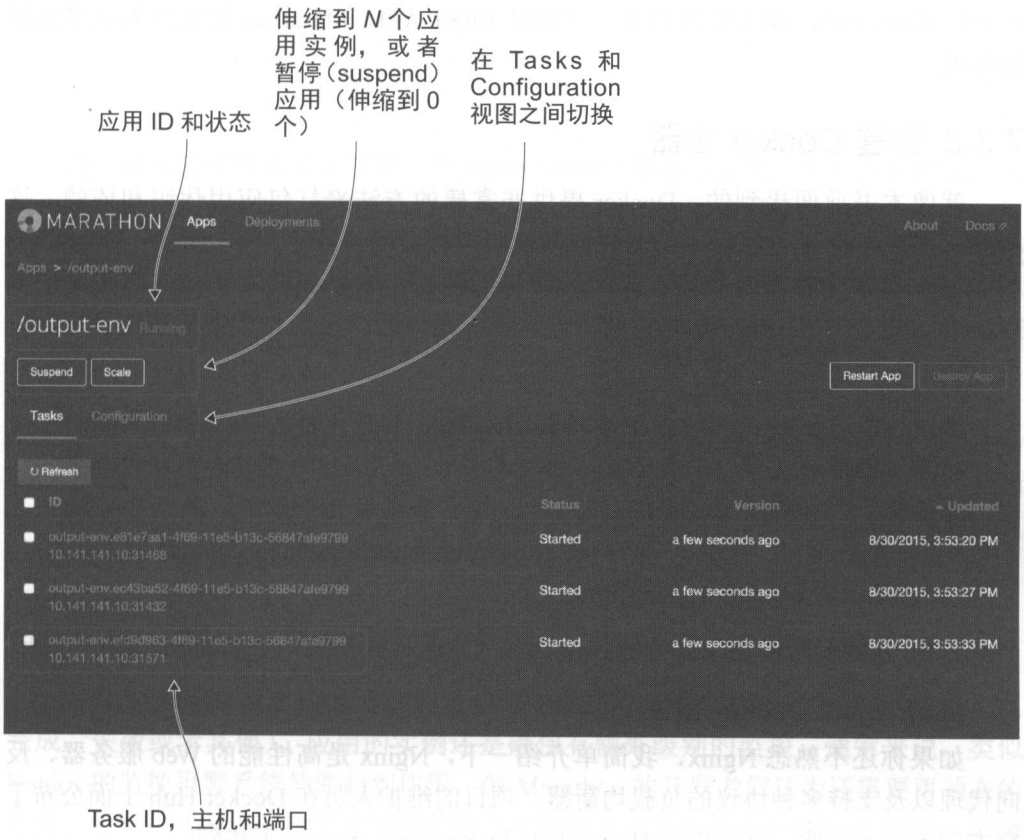


图 7.7 output-env 的 Marathon 应用细节和管理页面

就如前面这张图所示，你得到相当数量的关于运行中的应用的信息，同时也有一些操作例如伸缩应用实例、重新部署应用或者把它们全部销毁。通过单击 Scale 按钮，你可以输入特定应用的实例（Mesos 任务）数，允许你轻易地在负载升高时扩容，在负载降低时缩容。更进一步来说，在获知你的 HAProxy 负载均衡器配置会在每分钟动态生成和更新的情况下，你可以把你的监控系统接入 Marathon 的 API 来实现自动伸缩。

单击任意任务的 ID 会显示该任务的附加信息，如果单击主机名和端口会在浏览器中重定向到指定的应用实例（假设你的工作环境和 Mesos slave 上运行的任务的网络连通性是正常的）。

不幸的是，在写本书时，Marathon 的 Web 接口功能还是相当有限的，不能使用它去修改应用配置，执行更新回滚，或者部署 Docker 容器。但既然现在你已经见识过了通过 Web 接口怎么创建和部署应用（在本例中，通过执行 `bundle exec`

ruby app.rb)，那么让我们看一下通过 REST API，Marathon 能实现的更多高级操作吧。

7.3.2 部署 Docker 容器

就像本书前面提到的，Docker 提供非常棒的方法来打包应用代码和依赖，这样你可以在本地服务器上或云上的任何基础架构上面运行它。幸运的是，Marathon 和 Mesos 都有在容器内部运行应用和服务的能力。本书到此为止，你都需要依赖 Marathon 的 REST API 来部署它们。但这都不是问题，因为这个章节涵盖了这样做的方法。

> **提示**：Docker 镜像一般会推送到 Docker Hub 上面，它是 Docker 公司推出的基于网络的仓库。如果你更喜欢使用 Docker Registry 搭建私有的内部仓库的话，当你在部署 Docker 应用时，你需要再做几个步骤。关于 Docker Registry 的安装指引，请参考 https://mesosphere.github.io/marathon/docs/native-docker-private-registry.html。

首先，让我们看看使用官方的 Nginx Docker 镜像来部署一个新的 Marathon 应用。

部署 Nginx Docker 镜像

如果你还不熟悉 Nginx，我简单介绍一下，Nginx 是高性能的 Web 服务器、反向代理以及支持多种协议的负载均衡器。项目的维护人员在 Docker Hub 上面公布了官方的 Nginx 镜像，你可以在 Https://hub.docker.com/_/nginx/ 下找到。

在 Marathon 上面部署 Docker 镜像需要在定义你的应用时，在 JOSN 对象上面添加一个 container 字段。让我们看看如下列出的 docker-nginx 应用定义吧。

清单 7.1 在 Marathon 里部署 Nginx Docker 镜像

```
{
    "id": "docker-nginx",
    "instances": 1,
    "cpus": 0.5,
    "mem": 64.0,
    "container": {
        "type": "DOCKER",
        "docker": {
            "image": "nginx:1.9",
            "network": "BRIDGE",
            "portMappings": [
                {
                    "containerPort": 80,
                    "hostPort": 0
                }
```

Marathon 应用 ID

Docker 容器信息，指定镜像和网络模式

桥接 containerPort 80 到 Mesos 和 Marathon 使用的临时 hostPort

```
            ]
        }
    }
}
```

这个基本的应用定义了部署一个 Nginx Docker 容器的实例，并把容器内部的 80 端口（Nginx 监听端口）桥接到 Mesos 和 Marathon 提供的随机指派端口上。

假设上面的 JSON 对象被保存为 docker-nginx.json 文件，如下的 cURL 命令也可以启动 nginx：1.9 的 Docker 镜像为新的 Marathon 应用：

```
$ curl -H 'Content-Type: application/json' -d @docker-nginx.json
➥ http://marathon.example.com:8080/v2/apps
```

如果你使用 Marathon Web 接口浏览，或者查询 Marathon 的 /v2/apps/dockernginx 的 API 路径，你会获知应用的信息和每个正在运行的实例的信息，包括每个容器正在运行的主机名和端口。如果你浏览这些实例中的一个，你会看到默认的 Nginx 的欢迎页面。

7.3.3 执行健康检查和滚动应用更新

尽管 Mesos 能通过当前运行的进程来提供某个人物的状态（例如运行中，已完成，失效或者其他），应用的实例还是确保有额外级别的监控。通常来说，类似 Nagios 的监控报警系统就能起到作用，但 Marathon 的开发者们认为还需要更动态的东西。所以，Marathon 为指定应用的每个实例提供可选的基于 HTTP 或者 TCP 的健康检查。

当某个实例的健康检查开始失败——返回错误的 HTTP 代码或者 TCP 连接失败时，该任务会报告为不健康的。如果健康检查在特定的几次连续失败后，Marathon 会重启不健康的任务。所有这些健康检查的参数都是可配置的，下面我将简短介绍一下。

当你在执行应用或者服务的滚动升级时，这些健康检查也发挥着作用，它能确保到最小级别的服务或者称之为容量。这样，新的实例必须健康地启动后，升级才会继续执行下去。把这些特性与可动态配置的负载均衡器组合起来，Marathon 允许应用发布新版本时做到零停机部署。

让我们开始看一下健康检查是如何实现的，在本章节后面你将了解到更多滚动升级的知识。

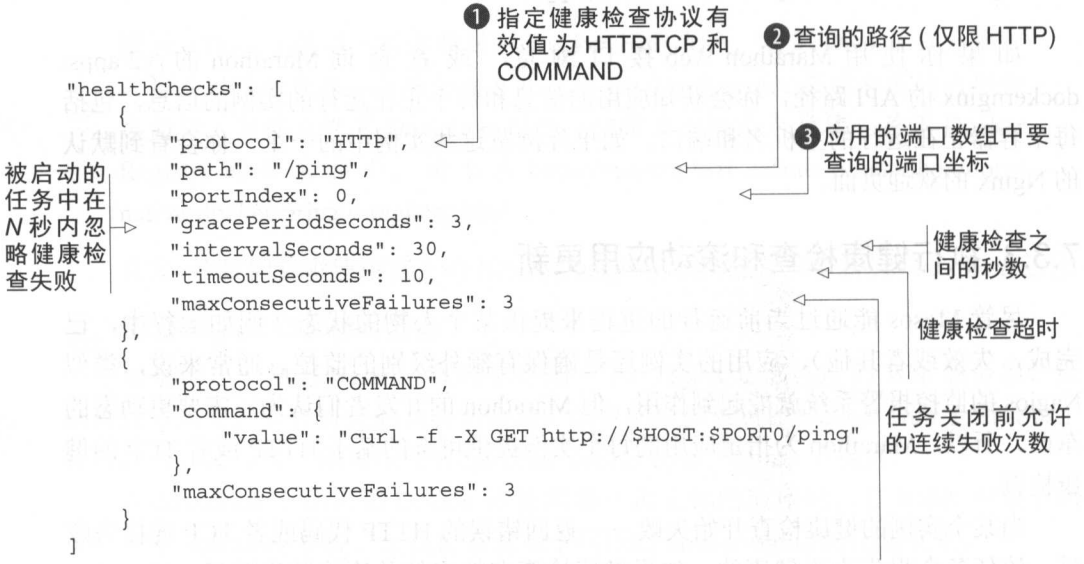
健康检查的剖析

任何 Marathon 应用都可以实现健康检查。在本书中，有如下三种方法，或者称之为协议是用来设置应用实例的健康检查的。

- HTTP——发起一个 7 层的 HTTP 请求到指定的端口和路径
- TCP——尝试与指定的端口进行 TCP 网络连接
- COMMAND——执行任意的命令来确定健康性（当前不适用于在 Docker 容器中运行的任务）

另外，你可以为健康检查定制化各种条件，例如 interval（检查间隔），grace period（任务启动初始可以忽略错误的时长），time-out（超时时间），连续失败的次数。让我们看看下面 Marathon 中应用健康检查的几个示例。

列表 7.2 Marathon 应用健康检查示例




```
"healthChecks": [
    {
        "protocol": "HTTP",
        "path": "/ping",
        "portIndex": 0,
        "gracePeriodSeconds": 3,
        "intervalSeconds": 30,
        "timeoutSeconds": 10,
        "maxConsecutiveFailures": 3
    },
    {
        "protocol": "COMMAND",
        "command": {
            "value": "curl -f -X GET http://$HOST:$PORT0/ping"
        },
        "maxConsecutiveFailures": 3
    }
]
```

这个例子指定了一个 HTTP 的健康检查 ❶，需要访问到指定的路径 ❷，取代了要重复定义应用端口，你可以使用 portIndex 字段来定义服务端口的坐标 ❸，在默认情况下，这个值为 0，代表数组中的第一个端口。

健康检查能有力地确保指定应用的大量实例或任务都启动成功，并且在应用层上是健康的。它在 Marathon 处理某个应用的滚动升级时扮演了决定性的角色——部署一个零停机时长的新版本应用。

处理基于健康的滚动升级

在升级期间，Marathon 默认会把新版本应用的所有配置任务都启动后，才会把旧版本的任务关掉。这样能确保在完全切换之前，新版本的所有实例都能运行并且保持健康状态。但就如 Marathon 大部分的事情里，这是可以高度定制的，取决于你们的团队、组织或者基础架构上的最佳策略。让我们通过一些示例来更深入了解升

级策略配置吧。

Marathon 应用中的升级策略，包含两个参数，最小健康容量（minimum health capacity）和最大替换容量（maximum over capacity），如下：

```
"upgradeStrategy": {
    "minimumHealthCapacity": 1.0,
    "maximumOverCapacity": 0.2
}
```

升级过程中保留在线的旧任务数的百分比

升级过程中额外启动的新任务数的百分比

因为这两个选项不是很直接明了，所以让我们假设一个示例应用的升级场景吧。某个指定的应用有 100 个为用户提供服务的实例（任务），现在你想要使用 Marathon 部署一个新的应用版本。如果设置最小监控容量（minimum health capacity）为 1.0(100%)，则意味着在升级过程中，Marathon 会保持 100 个实例。如果这个数值设得更低（比如 0.9 或者 90%），Marathon 会在已存在的任务中杀掉 10 个，为新任务的启动腾出空间。

另外一个值得关注的配置选项是最大替换容量（maximum over capacity），该选项允许 Marathon 在升级过程中，一次替换掉指定百分比的任务。在上面的例子中，Marathon 会启动 20 个新版本的任务，等到他们通过健康检查后，再关掉已经存在的旧任务。

这些数值可以被修改以满足你个人应用的 SLAs，或者根据集群中可用的资源进行微调。如果你的某个应用被特别授予了固定额度的资源，那么你可以设置 minimumHealthCapacity 为 0.9，maximumOverCapacity 为 0.0，这样应用在整个升级过程中都会保持 90% 的配置实例可用（也因此每次会替换 10 个实例）。

如我在章节前面提到的，引进了 Marathon's REST API 的话，那么你可以使用 HTTP 的 PUT 方法往 /v2/apps/<app-id> 这个路径发送请求，来对 Marathon 已有的应用进行配置更新。如果你有个叫做 test-app 的应用，并且这个 Marathon 应用的定义是保存在你的代码库中，或者以 marathon.json 的文件形式保存在磁盘上，那么你可以通过如下的 cURL 命令来升级该应用：

```
$ curl -H 'Content-Type: application/json' -X PUT -d @marathon.json
➥ http://marathon.example.com:8080/v2/apps/test-app
```

当滚动升级和通过 haproxy-marathon-bridge 或者 servicerouter.py 生成动态的 HAProxy 配置组合起来的时候，既会把新实例加入池中，又会把旧实例移除。整个升级过程至始至终，都能确保应用版本的无缝切换和用户的零停机体验。

如今你已经知道了部署简单应用、健康检查和滚动更新，让我们开始了解如何在一次 Marathon 单独部署中创建应用组和服务依赖。由于应用组是由独立的应用组

合起来的，本节中的很多内容也适用于下节。

7.4 创建应用组

7.3 节的示例已经展示了 Marathon 的威力，但你的应用有可能是由多个服务或者容器镜像组成的。基于这点，我将向你展示如何使用 Web 接口和 REST API 来部署组合多个独立的 Marathon 应用实例。当你以应用组的方式来定义你的应用以及它们的依赖服务时，Marathon 才真正展示它的实力。

7.4.1 理解应用组的构成

Maratho 的部署由单独的应用或者应用组的应用组成。而且，应用和应用组可以依赖于另外的应用或者应用组。在深入了解应用组的组成前，让我们看看如图 7.8 的应用组。

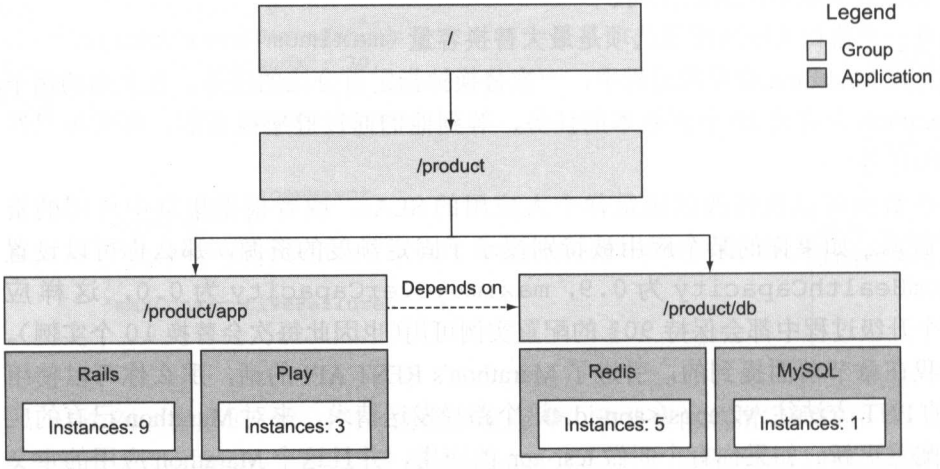


图 7.8 Marathon 应用组包含多个单独的应用

使用应用组，你只需要一个 JSON 对象就能创建复杂的应用和它们的依赖关系。它允许你轻易地定义依赖、部署升级以及伸缩整个运行在 Marathon 上的应用栈（可以是独立的应用，也可以是整个应用组）。现在你对应用组是如何组成有了更多的了解，让我们看看更多的现实场景吧。

7.4.2 部署应用组

为了最佳展示如何部署一个依赖数据库的应用，让我们使用本书的补充材料来部署一个示例应用（Keys and Values）。你可以通过图 7.9 看一下这个高级别的 Marathon 应用组。

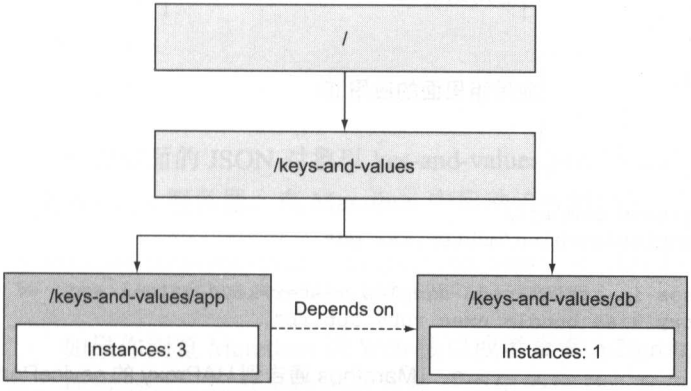


图 7.9 Keys and Values 应用部署的总览

Keys and Values 是一个使用 Sintra framework 编写的 Ruby Web 应用。它允许用户操作 Redis 这种 key/value 数据库的数据。你将运行多个应用实例和一个 Redis 实例，使用 HAProxy 来处理服务之间的通信。

> **注意**：就如本章节前面提到的 OutputEnv 的 app 一样，这里的 Keys and Values 也不是以 Docker 镜像的方式部署的。如果要部署这个 app，你必须在每个 Mesos slave 节点上安装 Ruby (1.9.3 版本或者更高) 和 Bundler。可以使用你的系统包管理器安装 Ruby，Budler 则可以通过运行命令行 `sudo gem install bundler` 来安装。这里有一点要说明的：Docker 能在没有系统管理员干涉的情况下，让发布应用和系统依赖库更为简单。

部署 Keys and Values 实例应用

列表 7.3 把你到目前为止了解到的如何创建一个叫做 keys-and-values 的 Marathon 应用组的知识都集合起来。在这个应用组中，Redis 在应用启动前就是可用的。你得注意必须在集群的每个机器上都运行 haproxy-marathon-bridge，这样应

用才能连接到数据库。

清单 7.3 在 Marathon 中部署 Keys and Values

```
{
    "id": "keys-and-values",
    "apps": [
        {
            "id": "app",
            "instances": 3,
            "cpus": 0.1,
            "mem": 128.0,
            "disk": 0.0,
            "uris": [
                "https://github.com/rji/
➥ mesos-in-action-code-samples/archive/master.tar.gz"
            ],
            "cmd": "cd mesos-in-action-code-samples-master/keys-values-app
➥ && bundle install --retry 3 && bundle exec ruby app.rb",
            "ports": [
                8080
            ],
            "env": {
                "REDIS_HOST": "localhost",
                "REDIS_PORT": "9000"
            },
            "dependencies": [
                "/keys-and-values/db"
            ]
        },
        {
            "id": "db",
            "instances": 1,
            "cpus": 0.1,
            "mem": 128.0,
            "disk": 0.0,
            "container": {
                "type": "DOCKER",
                "docker": {
                    "image": "redis:3.0.3",
                    "network": "BRIDGE",
                    "portMappings": [
                        {
                            "containerPort": 6379,
                            "hostPort": 0,
                            "servicePort": 9000,
                            "protocol": "tcp"
                        }
                    ]
                }
```

应用组的 ID

应用组里面的应用 ID

Marathos 通告到 HAProxy 的 sevicePort

应用的环境变量；在本地主机：9000 上连接到 Redis

确保数据库在 app 前是可用的

应用组内的数据库服务 ID

桥接容器内 6379 端口到 Marathos 的随机 hostPort

HAProxy 应该转发 9000 端口上的连接到这个 app

```
            },
            "healthChecks": [
                {
                    "protocol": "TCP",
                    "portIndex": 0,
                    "maxConsecutiveFailures": 3
                }
            ]
        }
    ]
}
```

在 Marathon hostPort 上建立一个 TCP 健康检查

假设前面的 JSON 对象以 kes-and-values.json 的文件保存，下面的 cURL 命令行将会随 Redis 服务器，在 Marathon 中启动该应用组：

```
$ curl -H 'Content-Type: application/json' -d @keys-and-values.json
➥ http://marathon.example.com:8080/v2/groups
```

如果你浏览 Marathon 的 Web 接口或者查询 /v2/groups/keys-and-values 的 API 路径，你将会获取到整个应用组和每个应用的信息。如果你浏览到其中的某个实例，就能够和这个应用进行交互。

另外，请注意 /keys-and-values/ 应用中指定的端口数组，该服务端口会被 haproxy-marathon-bridge 或者 servicerouter.py 使用来生成 HAProxy 的配置文件。如果你使用其中的一个脚本来架设负载均衡器，那么你可以通过访问 http://loadbalancer.example.com:8080 来访问应用。

7.5 日志和调试

尽管事实上 Marathon 是 Mesos 的一个 framework，但由 Mesos 提供的日志信息会伴随着 Mesos API 而终止。除了资源提供、任务启动和 framework 注册与再注册外，Mesos master 不会了解 Marathon 发生的大量信息。因此，由 Mesos 本地应用和数据中心其他运行的应用提供的日志同样重要。

由 Marathon 提供的日志信息可以帮助问题的排查，或者观察服务本身的常见行为。这些日志信息包括 API 路径访问、资源提供的接受与拒绝、任务状态的升级，以及应用的伸缩。你将在与 Marathon 服务的日常交互中观察到这些信息。

本节将涵盖 Marathon 如何处理日志，以及可用的配置。

7.5.1 配置 Marathon 日志

默认地，Marathon 的启动脚本能使用 logger 的接口来增加写到系统日志的条

目。默认情况下，会记录 INFO 级别的日志，提供相当数量的关于 API 路径被访问和 Marathon 应用部署状态的信息。和其他大多数配置一样，日志的级别也是可以配置的，并且 Rsyslog 可以被定制化来更好地适配环境。让我们看一下修改每样东西的要求。

修改日志级别

Maratho 提供了一个配置选项来修改应用的日志级别。你能使用 --logging_level 的配置选项（或者通过创建一个带值的 /etc/marathon/conf/logging_level 文件），将日志级别调整到如下级别，严重性依次递增，all，trace，debug，info，warn，error，fatal 和 off。默认情况下，Marathon 发布的时候，日志级别为 info。

从 syslog 重定向 Marathon 的日志到专用的日志文件

默认情况下，Marathon 将定义级别以上的日志传到系统日志上。对于某些人来说，他们拥有一个令人满意的日志集中平台，如 Logstash 或者 Splunk 能够轻易地消费系统日志条目并且将它们重构成更轻易查询的数据。对于其他人而言，遍历 /var/log/messages 或者 /var/log/syslog（取决于发布版本）的系统日志文件，筛选出 Marathon 的日志信息来排查某个问题是非常乏味的。

如果你还没有致力于日志管理计划的话，或者你比较喜欢筛选 Rsyslog 并且把 Marathon 的日志条目写在属于它自己的日志里，那么你可以按下面的内容创建 /etc/rsyslog.d/10-marathon.conf 文件：

```
if $programname == "marathon" then {
    action(type="omfile" file="/var/log/marathon.log")
}
```

提示：如果想了解更多关于 Rsyslog 的可用筛选条件的话，可以查看官方的文档：www.rsyslog.com/doc/v8-stable/configuration/filters.html。

像往常一样，如果你计划把日志写到磁盘上的某个专用文件上，记得考虑创建一些日志轮替规则，来确保这些日志文件不会增长得太大以至于撑满系统的日志分区。

7.5.2 调试 Marathon 应用和任务

因为 Marathon 的应用实例通常都是长时间运行的 Mesos 任务，所以大部分的日志和调试内容已经在前面第 5 章中提到了。快速地再回看一次很重要，同时也添加了一些调试 Docker 容器的信息。

复习调试 Mesos 任务

每个 Mesos 集群中运行的任务在 Mesos 的 Web 接口 http://mesos-master.example.com:5050. 是可见的。如果导航到一个非主 master 节点，会自动重定向到 leader 节点上。从这里，你可以查看某个运行中任务的沙盒。在沙盒中，两个文件 (stdout 和 stderr) 会被自动创建，来捕捉该任务任何控制台的输出。

进程（任务）会持续运行直到它被关掉、丢失或者退出，此时 Mesos 的任务状态也会更新。如果某个任务丢失了，意味着 Marathon 应用并不是运行在 100% 的负载上，Marathon 会无条件地自动重启失败的任务。

在 Marathon 中，应用实例既能以 shell 命令行（例如 bundle exec ruby myapp.rb）的方式，又能以 Docker 容器（通过在应用的 JSON 对象中添加 container 部分）的方式来定义。这两种方法有个重要的区别：单独的命令行将会使用 Mesos 命令行执行器来启动，而 Docker 镜像会使用内置的 Docker 执行器来启动。由于这个区别，所以在 Mesos 集群里启动 Docker 容器时，经常会有这种典型的失败模式：失败的任务沙盒中，日志文件通常都是空的。

调试使用 Marathon 和 Mesos 启动的 Docker 容器

如果 Docker 容器的应用被配置成记录到标准输出和标准错误上的话，那么日志条目信息会分别出现在 Mesos 沙盒里的 stdout 和 stderr 文件上。这种在 Mesos 集群上运行的任务是非常典型的。然而有时候，通过 Marathon，某个 Docker 容器也会启动失败，那么沙盒中的日志文件也将是完全空白的。发生了什么呢？那让我们看看这种情况吧！

让我们假设你创建和部署了一个叫做 docker-invalid-container-example 的 Marathon 应用，它被配置成从 fake-org/thisisnotmybeautifulcontainer 镜像下载并运行 Docker 容器。唯一的问题是，这个镜像是不存在的。当你观察到你的应用并没有启动失败，那么导航到 Mesos UI 时想去搞清楚哪里出了问题时，你会发现 stdout 和 stderr 日志文件都完全是空白的。

当你往更深一层查看时，在 Marathon 日志里，你会发现不太明显的这么一条日志条目：

```
Oct 18 21:24:48 mesos marathon[1335]: [2015-10-18 21:24:48,434] INFO
Received status update for task docker-invalid-container-example.a753e2af-
75de-11e5-a1ac-56847afe9799: TASK_FAILED (Abnormal executor termination)
(mesosphere.marathon.MarathonScheduler$$EnhancerByGuice$$417430f8:100)
```

不幸的是，除了本来就知道的任务失败了，你并没有获知到更多的信息。在这个案例中，真正的错误原因被记录在试图启动 Docker 容器的机器上面的 /var/log/mesos/mesosslave.INFO 文件里。如果更进一步调查，你会发现如下的错误信息：

```
E1018 21:24:48.423717 18610 slave.cpp:3112] Container '978cc82b-6838-4c2a-
8487-3516357a8641' for executor 'docker-invalid-container-example.a753e2af-
75de-11e5-a1ac-56847afe9799' of framework '20150930-024708-16842879-5050-
1180-0000' failed to start: Failed to 'docker pull fake-
org/thisisnotmybeautifulcontainer:latest': exit status = exited with status
1 stderr = Error: image fake-org/thisisnotmybeautifulcontainer:latest not
found
```

啊哈！沙盒日志为空的原因是执行器启动失败了（因为 Docker 镜像不存在），这意味着没有日志条目会写到沙盒里。修正应用定义中的错误配置可以解决这个问题。

这个场景恰恰说明了一个如 Elasticsearch、Logstash 和 Kibana (ELK) 的开源栈或者如 Splunk 的商用产品的日志集中方案对基础架构来说，是有价值的附加品。相比于查出 Mesos 哪个 Mesos slave 启动容器失败了，然后再登录到主机上面翻看日志这种烦琐的过程，你只需要在一个接口上就能轻易地完成搜索。如果想了解 Logstash 的更多信息，你可以查看 www.elastic.co/guide/en/logstash/current/introduction.html 的官方文档。如果想了解 Splunk 的更多信息，你可以查看 www.splunk.com/en_us/products/splunk-enterprise.html。

现在如果你对 Docker 容器的日志进行更多的控制，或者想在 Mesos slave 上面使用 docker logs 命令，我建议你看一下下面的文档。

- 配置 Docker 日志驱动 C: https://docs.docker.com/reference/logging/overview/。
- 通过 Marathon 传递任意的 Docker 参数：http://mesosphere.github.io/marathon/docs/native-docker.html#privileged-mode-and-arbitrary-dockeroptions。

刚刚我只是提及了记录和观察 Docker 容器状态的冰山一角。还有别的选项，例如使用 JSON 结构化日志，或者把 Docker 容器的日志传送到系统日志里面。除了阅读 Docker 的官方文档外，相信你也会对 Jeff Nickoloff 编写的《Docker 实战》（Manning，2016）一书感兴趣的。

7.6 小结

在本章中，你学到了在 Marathon 上部署服务和应用。本章涵盖的主题包括安装、配置、应用部署、应用依赖，以及服务发现和路由。下面是要记住的几点：

- Marathon 是一个 Mesos 私有 PaaS 平台，它以 Mesos 任务的方式部署长时间运行的应用或者服务，并且当任务失败时能够自动重启。通常来说，Marathon 之于 Mesos，等价于 init 系统之于传统的 Linux 操作系统。
- 能够使用 HAProxy 加上 haproxy-marathon-bridge 或 servicerouter.py 脚本，或者 Mesos-DNS 来处理服务发现和路由。

- 应用部署可以由单独的应用来组成，也可以由应用组里的应用来组成。应用组既可以包含应用，也可以包含其他的应用组。
- 健康检查可以使用 HTTP、TCP 或者命令行的方式来实现，这确保了应用正在启动并且是有响应的。对于每个应用，也可以配置允许连续失败的次数、最大超时时间和检查的间隔时长。
- 为了确保在不中断用户的情况下，应用的新版本可以滚动发布，Marathon 允许使用应用的 JSON 对象中 `minimumHealthCapacity` 字段对现有应用进行滚动升级。配合健康检查，Marathon 能确保在关闭旧版应用前先启用新版本应用，以继续正常提供服务。
- 通过应用组，Marathon 应用除了最小的健康要求外，还能拥有对其他应用或者服务的依赖性（例如一个 Rails 应用取决于一个数据库需要的确切的实例数）。

尽管本章提供了几个有深度的例子，但比起我在本书中提到的，Marathon 拥有更多的特性，而且还会在每个版本中添加越来越多的特性。最新的文档，可以在如下地址查看：https://mesosphere.github.io/marathon/docs/。

在第 8 章，你将了解到用来执行调度任务的热门 Chronos framework。它通常也被认为是 Mesos 集群上的 Cron 作业。

8 使用Chronos管理计划任务

本章内容

- 安装和配置 Chronos；
- 创建计划任务；
- 任务失败时查看输出和报警。

在传统的 Linux 操作系统里面，Cron 守护进程会负责基于时间的命令或者脚本，统称为 *Cron jobs* 的执行。如果你把 Mesos 当作“分布式的操作系统内核”的话，那么 Chronos 则如项目网页所描述的一样，它相当于 Mesos 的 Cron 系统：能处理 Mesos 集群上基于时间的调度作业（任务）。如果你想在 Mesos 集群上运行任意的命令，你需要指定计划，以及作业所需要的 CPU 和内存资源。

本章将为你介绍 Chronos 这个热门、开源的 Mesos framework，最开始它是由 Airbnb 开发用来处理其复杂的数据分析管道的。Chronos 被设计为高可用的，当作业失败时会自动重试，并且尽可能的灵活。所以，它是 Mesos 的 Cron 守护进程，而非简单地用来处理数据分析。

Chronos 能用来处理计划调度命令和脚本，并使用 Mesos 内置的容器。通过 Linux 的控制组（cgroups）和 Docker 容器，它能开箱即用地运行命令。使用

Chronos 的特性集，你能轻易并可靠地创建单独的基于计划的作业，同时也能通过简单地指定计划和作业所需的资源（由 Mesos slave 提供）来处理复杂的基于依赖的作业和管道。这帮你确保了基于时间的作业准时运行，同时也继续尽可能高效地使用数据中心的资源。

8.1 了解Chronos

到目前为止，你已经通过运行多种不同的 framework 的情况探索了 Mesos，并且学习了在不用担心管理数量众多且静态配置的机器的情况下，如何达成数据中心的更高利用率。但通过 Chronos，我将要稍微改变这个方法来讨论 Mesos 集群上运行的计划任务是如何成就一个更弹性更准时的作业执行系统的。事实上，部署 Chronos 作为数据中心某台单独服务器上的 Cron job 强有力的替代品，是我在刚开始使用 Mesos 时做的第一件事情。借助于图 8.1，让我们来考虑在单独的机器上运行 Cron jobs 和在 Mesos 集群上运行它们的区别。

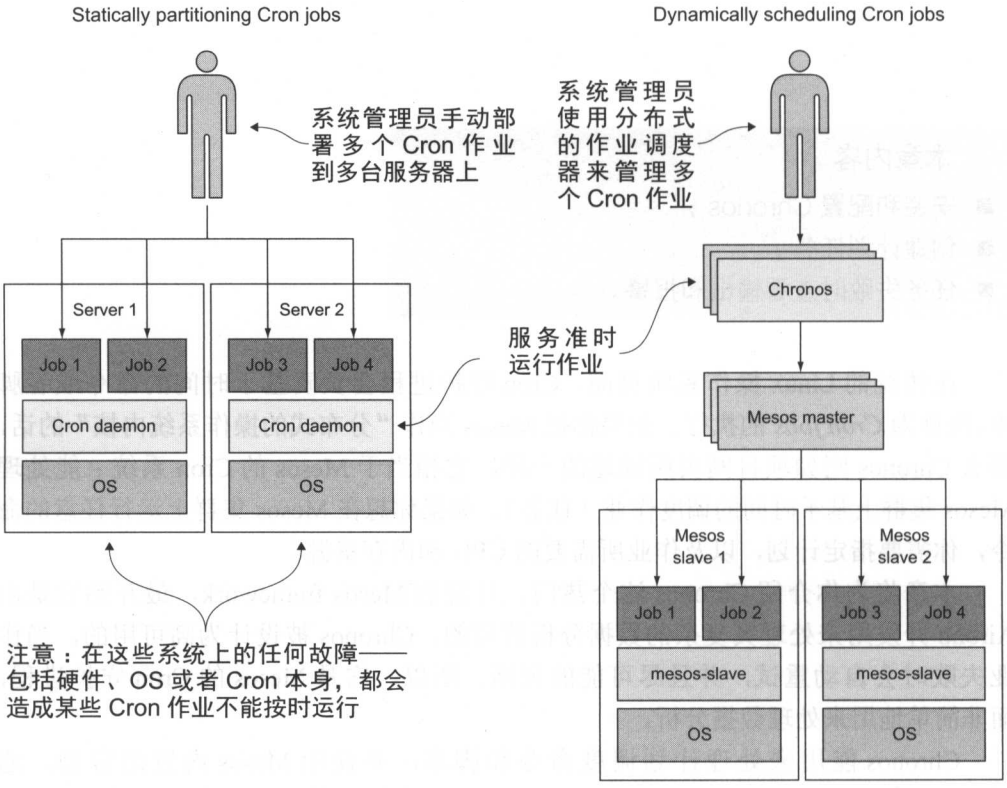


图 8.1 Mesos 和 Chronos 提供一个动态且容错的环境来执行基于时间的作业

我已经描绘了在传统情况下，系统管理员是如何部署计划作业，以及如何通过使用 Chronos 和 Mesos 来做一样的事情的。当然，你也可以部署几台新的虚拟机并且在上面创建 Cron 作业。但在一天的最后，你不需要关注哪个任务在哪个服务器上执行。相反，你只需要关注作业所需要运行的 CPU、内存和磁盘资源。

Chronos 允许你通过使用 Mesos 集群上的可用资源来调度任务。它的可容错和使用 Mesos 集群上的可用资源意味着你不需要担心硬件、OS 和 Cron 守护进程本身的错误异常。同时它也是高可用的，意味着你不需要担心 Chronos framework 中的某个实例失效，因为新的实例会被选举为 leader，可能你和你的用户甚至不会察觉到已经发生故障转移。

结合 Mesos 基于容器来实现资源隔离的方法（第 4 章已将提到过），每个作业都能在 Mesos 集群上的其他作业或者应用旁边不受干扰地运行。

注意：本章提到的 Chronos 版本为 2.4.0。

Chronos 在 Cron 守护进程的基本特性（指定计划和用户执行的命令）之上还包含大量的特性。这些特性允许你做下面的事情：

- 按计划执行作业；
- 在作业之间创建复杂的父子依赖关系；
- 在 Docker 容器中运行命令；
- 自动重试失败的作业。

在你深入部署 Chronos 和创建计划作业前，8.1.1 节先带你浏览 Chronos 的 Web 接口和 REST API。

8.1.1 探索 Chronos 的 Web 接口和 API

Chronos 有丰富的 Web 接口来管理计划作业并且收集相关的信息，这些都依赖于底层的 REST API。这使你能够在版本控制中以 JSON 对象的方式来保存 Chronos 作业的配置，并且使用 CI 系统（例如 Jenkins 或者 TeamCity）来部署变更。本节的稍后你将学习到 REST API，让我们先开始看看 Web 接口吧。

探索 Web 接口

和 Marathon（已在第 7 章讲解过）类似，Chronos 提供一个 Web 接口来管理计划作业并显示相关的信息。有些信息包含作业的状态（成功 / 失败），执行时长，以及作业的关系图。让我们快速浏览一下图 8.2 的 Web 接口的主页吧。

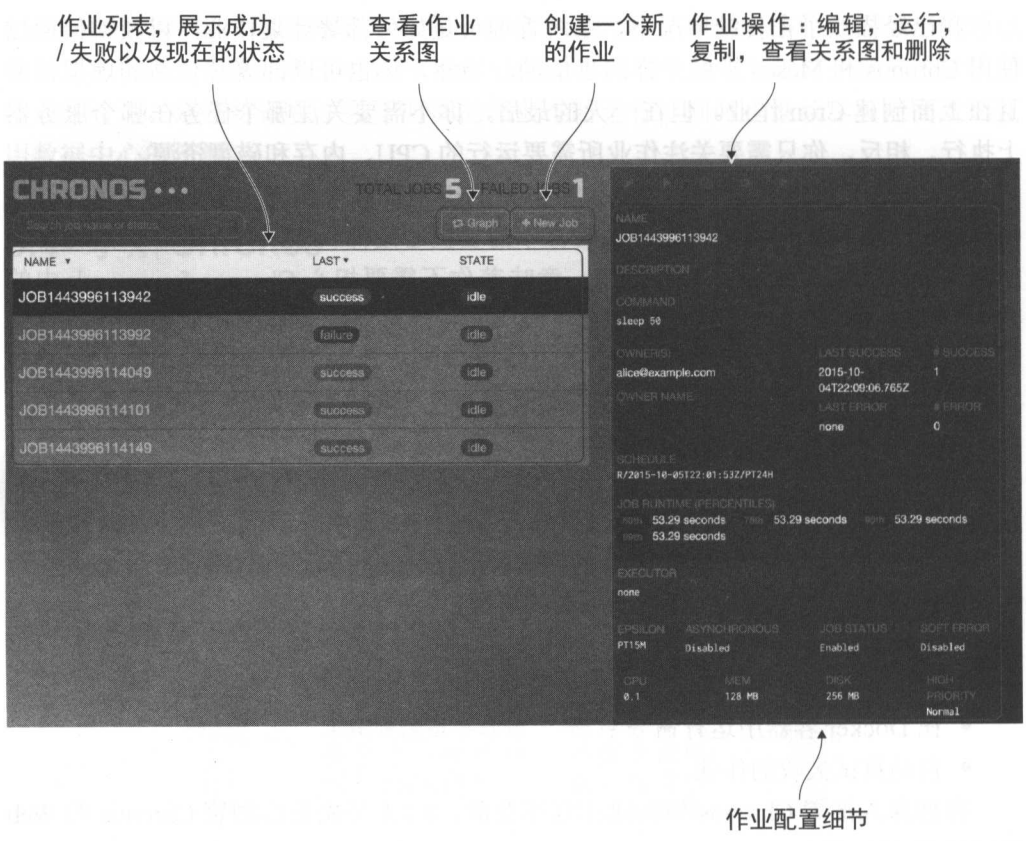


图 8.2 Chronos Web 接口总览

如你所见，Web 接口提供如下几个功能：

- 创建、配置和删除作业；
- 查看已存在的作业、当前的状态以及最后一次运行的情况；
- 通过内置的关系图查看作业之间的依赖关系。

尽管 Web 接口在创建普通任务和查看它们的依赖性上面很有帮助，但仍然有一些局限性：（目前）你不能使用 Web 接口来启动 Docker 容器，也不能使用更多高级的特性，例如使用 Mesos 提取器来下载文件到沙盒中，指定时区或者设置环境变量。基于这个原因，本章将更详细地来讲解 API。

探索 REST API

Chronos 基于 JSON 的 REST API 接口让创建、更新或者移除作业都很简单。首先，表 8.1 列举了一些重要的 Chronos API 路径和操作它们的 HTTP 方法。因为基于使用的 HTTP 方法不同，每个路径的调用结果是不同的，所以表格包含了每个 API 路径

的多种行为，取决于括号内的 HTTP 方法。贯穿全章，你将使用这些路径来创建越来越复杂的作业。

表 8.1 重要的 Chronos API 路径

API 路径	描　述
/scheduler/jobs	查询一个 Chronos 实例上所有的作业 (GET)
/scheduler/iso8601	创建 (POST) 或者修改 (PUT) 计划上运行的一个作业
/scheduler/dependency	在父作业完成后创建 (POST) 或者修改 (PUT) 一个依赖的作业
/scheduler/job/<job-id>	对已有的一个作业进行操作：手动运行（PUT）或者删除（DELETE）

Chronos REST API 的完整文档，包含了本文中没有提到的额外特性，能够在互联网上的 http://mesos.github.io/chronos/docs/api.html 地址查询到。

> **提示：**在本章所有基于 API 的例子中，你会了解如何直接提交 JSON 到 API 中。如果你对更高级别的实现有兴趣，一个 Chronos 的开源 Python 客户端库覆盖各种 API 路径和在它们上面操作所需的 HTTP 方法。更多信息，请查阅 http://github.com/asher/chronos-python。

你将在本章后面学到使用 Web 接口和 REST API 来创建任务。现在让我们继续看如何部署 Chronos。

8.2 安装并配置Chronos

前面的章节给你介绍了 Chronos 项目以及简单地总览了它的 Web 接口和 REST API。现在，让我们准备好 Mesos 集群通过安装和配置 Chronos 来处理计划好的作业。

8.2.1 先决条件的检验

要想在基础架构中部署 Chronos，有如下几个先决条件需要满足。对于 2.4.0 版本（以及本书）来说，依赖条件如下：

- Java 1.7 或更高版本；
- Mesos 0.22.2 或更高版本；
- ZooKeeper。

为了让 Chronos 实现高可用，你需要像在第 3 章中部署奇数个 Mesos master 一样，部署多个 Chronos 节点。在大部分场景下，部署 3 个 Chronos 实例是合适的。并且每个实例都会使用 ZooKeeper 集群来实现 leader 选举和状态维持。

为了更简便地部署，可以考虑把 Chronos 部署在与 Mesos master 相同的机器上面，

或者把 Chronos 当成 Marathon 上的一个应用。为了让本节的例子更容易理解，让我们假设在第 3 章中你已经部署好了 Mesos master，你把 Chronos 安装在与其相同的机器上。

Chronos 独立的 DNS 域名

在一个高可用的 Chronos 部署中，多个实例被部署在多个主机上。基于管理的意图，你会考虑创建一个独立的 DNS 记录指向到你的 Chronos 集群。幸运的是，Chronos 有这么一个特性：无论请求从哪里来，一旦被发送到非主节点，该请求都会被透明地代理到 leader 节点进行处理。

你可以使用两种独立域名的方法来连接 Chronos 集群：

- DNS 负载均衡——通过多个指向每个 Chronos 实例的 A 或者 CNAME 记录创建独立的 DNS 域名。
- HTTP 负载均衡——使用负载均衡器（例如 HAProxy）来转发 Chronos 的连接，创建一个 DNS 域名指向该负载均衡。

每个选项都有优点和缺点。举例来说，DNS 负载均衡便于配置，但如果 Chronos 其中的一个实例不可用，可能会造成失败的连接尝试。另外，添加一个 HAProxy 负载均衡器，可以自动地移除后端池中失效的实例，但需要在客户端和 Chronos 之间添加该服务。

接下来，让我们继续安装和配置 Chronos 吧。

8.2.2 安装 Chronos

安装 Chronos 最简单的方法是使用第 3 章中安装 Mesos 期间配置的 Mesosphere 安装包仓库。根据你的 Linux 发行版本，运行下面的命令来安装 Chronos：

- RHEL 和 CentOS:

```
$ sudo yum install chronos-2.4.0-0.1.20151007110204.el7
```

- Ubuntu:

```
$ sudo apt-get install chronos= 2.4.0-0.1.20151007110204.ubuntu1404
```

尽管可以从源码来下载和安装 Chronos，但这个过程比使用由 Mesosphere 提供的安装包更复杂。所以，本书不会涉及编译指引。但如果你有兴趣的话，可以查看项目的“Getting Started”文档 (http://mesos.github.io/chronos/docs/getting-started.html) 来获取最新的指引。

8.2.3 配置 Chronos

和 Marathon 类似，Chronos 也有一些配置规范，这取决于你的部署方式以及偏好。因为 Mesosphere 持续开发 Chronos 并为社区提供了安装包，所以其基于文件的配置（类似于 Mesosphere 的 Mesos 和 Marathon 安装包）可能是上手 Chronos 最简单和直截了当的方式，你可能需要在操作系统上创建 /etc/chronos/conf/ 目录。

Chronos 有多种配置选项，如 http://mesos.github.io/chronos/docs/configuration.html 所指定的。表 8.2 是你在部署 Chronos 时，我找出来的最值得一提的具体配置选项。

表 8.2 重要的 Marathon 配置选项

配置选项	描　述
master	Mesos master 所使用的 ZooKeeper URL。如果存在 /etc/mesos/zk(Chronos 部署在 Mesos master 上），那么该文件的值会被自动使用
zk_hosts	Chronos 用于 leader 选举和状态的以分号分隔的 ZooKeeper 主机列表。如果存在 /etc/mesos/zk(Chronos 部署在 Mesos master 上），Chronos 会使用和 Mesos 一样的 ZK 主机和端口，但是会创建自己的 /chronos/state znode 节点
hostname	指定 Chronos 节点的 DNS 域名或者 IP 地址
mesos_role	如果设置，以指定的角色将 Chronos 注册到 Mesos 集群中（详见第 6 章）
mesos_authentication_principal	主要用于 framework 认证的 Mesos（详见第 6 章）
mesos_authentication_secret_file	framework 认证时包含密钥的文件路径（详见第 6 章）
mail_from	Chronos 发送邮件通知时的发件人地址
mail_server	发送邮件的 SMTP 服务器。如果服务器需要认证，则需组合 mail_user 和 mail_password，如果需要加密，还需要加上 mail_ssl 配置选项
slack_url	发送通知到 Slack 的 Webhook URL

另外，Chronos 也支持简单认证和 SSL，我特意把这个特性从上表中抽离，是因为有些用户可能更喜欢在多个 Chornos 实例前端的负载均衡器设置认证和 SSL 终端。如果你对项目配置文档中的 ssl_keystore_path 和 ssl_keystore_password 选项感兴趣的话，请看 http://mesos.github.io/chronos/docs/configuration.html。

在每个 Mesos master 节点安装和配置 Chronos 且服务启动完成后，你可以通过 http://chronos.example.com:4400 访问 Chronos 的 Web 接口。

8.3 使用简单的作业来工作

本章把 Chronos 作业分成两类：简单的和复杂的。首先让我们搞清楚：

- 在本章中，简单的作业是指一个独立的基于计划的 Chronos 作业。该作业和其他的 Chronos 作业并没任何父依赖关系或者子依赖关系。它类似于每天发送一封 E-mail 邮件，或者每晚执行一个数据库备份。
- 从 8.4 节开始，复杂的作业是指至少一个基于计划的 Chronos 作业，还伴随这一个或者多个基于依赖的作业（该作业只有在其父作业成功完成后才能运行）。这类似于一组用于数据分析的 ETL 管道，可以在每小时、每天或者每周执行。

你将从一个独立的作业开始，类似于查看传统的 Cron 守护进程一样的配置来上手 Chronos。

8.3.1 创建基于计划的作业

让我们面对这种事实：在每个组织内都会有按计划运行某种任务的需求。不管是每天晚上传上亿的数据到支付服务供应商还是每小时更迭日志文件，又或者是每半小时运行一个 ETL 管道，这些都是非常清晰的用例——按预定计划运行一个任务，并且希望它是可靠并且准时的。

和我们大多数人都很熟悉的传统 Cron 守护进程比较，Chronos 允许你在 Mesos 集群中调度资源并且在容器内运行一个计划好的作业。通过手动供给资源给独立的机器（可能某个节点也会有异常）来运行 Cron 作业会浪费时间和资源，而 Mesos 和 Chronos 通过集群内的可用资源提供了可靠的启动作业的方法，并且在它们失败的时候还会自动重新尝试运行。

在开始创建第一个 Chronos 作业前，让我们先看看传统 Cron 守护进程的局限性。

传统 Cron 作业的剖析

下面的例子展示了 Cron 中基于计划的作业的最基本形式。每分钟 alice 用户 sleep 30s。

```
* * * * * alice /bin/sh -c 'sleep 30'
```

这应该是非常熟悉的例子，它通常会被部署在一台单独的机器上，并且会在机器以及 Cron 守护进程存活且健康的情况下一直执行下去。不管这个例子有多么简单，我们来思考一下 Cron 系统的缺陷：

- 作业必须在运行了 Cron 守护进程的指定系统上进行手动部署。
- 作业无法以多于一分钟一次的频率运行。

- 作业对其他的作业没有父子依赖关系。
- 标准的 Cron 语法对某些人来说不够直观。

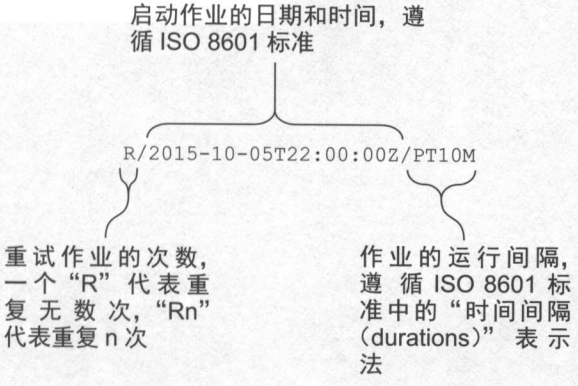
当然，你可以说像 Anacron 的工具允许你在操作系统经历宕机的时候，还能调度错过的作业。同时，你也能通过执行一条命令来处理多个作业之间的关系。但最终，你还得处理数据中心的其他 Cron 作业所在的机器。

基于计划的 Chronos 作业的剖析

相反，在 Chronos 中基于计划的作业可以用 JSON 对象展示，最简的方式如下所示：

- 计划；
- 名字（也指 ID）；
- 命令。

作业的计划由三部分组成，每部分用“/”分隔：



在熟悉了 Chronos 作业的计划格式后，你能很容易理解作业开始的日期和时间、之前的执行频率以及重复的次数（包括无限次）。上面例子中的作业从 2015 年 10 月 5 日晚上 10 点（协调世界时 UTC）开始每 10 分钟执行一次，并且重复无数次。

提示：关于 ISO 8601 日期/时间标准的更多信息，可以浏览 https://en.wikipedia.org/wiki/ISO_8601。

现在你已经看到 Chronos 和传统的 Cron 守护进程的不同之处，也了解了它的计划格式。让我们开始学习如何通过 Chronos 的 Web 接口来创建一个简单的基于计划的作业。

使用 Web 接口创建基于计划的作业

在创建作业时，Web UI 提供了各种不同的字段，包括名字、描述、命令行以及

计划。大部分的字段都是一目了然的。在图 8.3 中，你可以创建一个简单任务：睡眠 60 秒然后退出。

点击作业的高级选项链接可以让你设置附加的选项，例如 epsilon（Chronos 等待从 Mesos 获得资源提供的时间）以及运行作业所需要的资源（CPUs、内存、磁盘等）。在目前为止，默认的选项都是适宜的，你不需要修改这些选项。出于完整性的考虑，我在这里把它们都展示出来。

不幸的是，Web 接口并没有提供 Chronos 中所有可用的作业选项。为了能使用更多的高级特性，你需要转而运用 REST API。

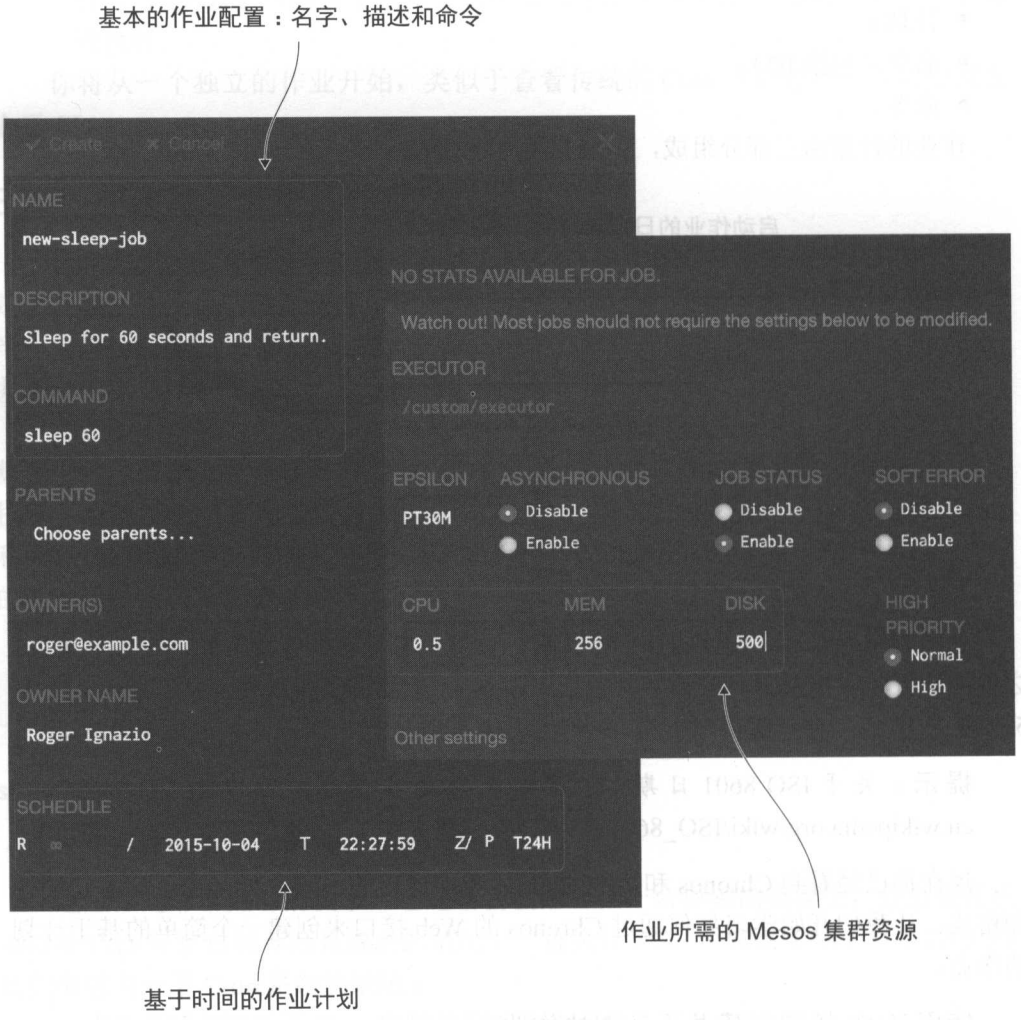


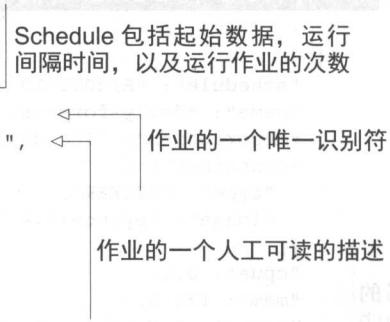
图 8.3 通过 Web 接口定义 Chronos 作业的配置

通过 REST API 创建基于计划的作业

REST API 中包含了 Web UI 中没有的特性，例如 Docker 容器的信息、时区的支持，以及使用 Mesos 提取器将文件下载到任务的沙盒中的功能。下面的列表展示了用安装 REST API 实现如同图 8.3 中的作业。

清单 8.1 Chronos 中的简单 Sleep 作业

```
{
  "schedule": "R/2015-10-04T22:57:59Z/PT24H",
  "name": "new-sleep-job",
  "description": "Sleep for 60 seconds and return.",
  "cpus": 0.5,
  "mem": 256,
  "disk": 500,
  "command": "sleep 60"
}
```



假设上面的 JSON 以 simple-sleep.json 的文件保存，下面的 cURL 命令行将在 Chronos 中创建一个 new-sleep-job 的作业：

```
$ curl -H 'Content-Type: application/json' -d @simple-sleep.json
➥ http://chronos.example.com:4400/scheduler/iso8601
```

如果此时你浏览 Chronos 的 Web 接口，你将看到该作业的信息，包含它最后运行的结果、现在的状态以及历史的运行时长。

8.3.2 使用 Docker 创建基于计划的作业

如本书第 1 部分提到的，使用 Docker 容器能帮忙打包你的代码和其依赖的运行环境，然后以一个独立的制品发布到集群的任意节点上。以往，你需要提交变更请求，在作业可能运行的每个节点上安装依赖包，而现在你能够通过构建并发布 Docker 镜像来取代前面的做法。

因为这种可移植性是有价值的，Chronos 的维护人员内置了启动 Docker 容器作为计划好的作业的支持。这意味着启动 Docker 容器来执行一条命令（使用 CMD）并退出，或者在容器内部运行一条命令（使用 ENTRYPOINT）。

这里有一个非常有用的特性，Mesos 的任务沙盒目录会自动映射到容器内部的路径，所以你可以继续使用 Mesos 的特性，例如使用提取器去下载你计划好的作业中需要的任意文件或者脚本。让我们看一下 Mesos 提取器是怎样下载一个脚本到沙盒中的，然后在附带 Python 3 的 Docker 容器中执行该脚本。

使用 REST API 创建作业

使用 Cron（或者 Chronos）的一个通常用例是在一定时间内自动发送 E-mail。本书的补充材料包含有一个示例脚本：email-weather-forecast.py 。这个脚本会根据用户的指定邮编对应地区来电邮他们最新的天气预报。在以下的列表中，让我们看一下如何通过 Chronos 的 REST API 来部署这个作业。

清单 8.2 在 Docker 容器中执行脚本

```
{
  "schedule": "R/2015-10-28T00:00:00.000Z/PT24H",
  "name": "daily-forecast-97201",
  "description": "The daily NWS weather forecast for Portland, OR",
  "container": {
    "type": "DOCKER",
    "image": "python:3.4.3"
  },
  "cpus": 0.1,
  "mem": 128.0,
  "owner": "user@example.com",
  "uris": [
    "https://raw.githubusercontent.com/rji/
    ➥ mesos-in-action-code-samples/master/email-weather-forecast.py"
  ],
  "command": "cd $MESOS_SANDBOX && python3 email-weather-forecast.py",
  "environmentVariables": [
    { "name": "TO_EMAIL_ADDR",   "value": "user@example.com"    },
    { "name": "FROM_EMAIL_ADDR", "value": "weather@example.com" },
    { "name": "ZIP_CODE",        "value": "97201"               },
    { "name": "MAIL_SERVER",     "value": "mail.example.com:25" },
    { "name": "MAIL_USERNAME",   "value": "weather@example.com" },
    { "name": "MAIL_PASSWORD",   "value": "ItsTopSecret!"       }
  ]
}
```

容器信息；此处我们使用 Docker 容器 python:3.4.3

从本书的 GitHub 库中下载这个例子

如果作业失败，Chronos 会发 E-mail 到 owner

转换到沙盒目录中并运行脚本

指定脚本所用的环境变量

假设上面代码列表的 JSON 以 simple-docker.json 的文件保存，下面的 cURL 命令行将在 Chronos 中创建一个 daily-forecast-97201 的作业：

```
$ curl -H 'Content-Type: application/json' -d @simple-docker.json
➥ http://chronos.example.com:4400/scheduler/iso8601
```

提示：你可以通过以 HTTP PUT 方法而非 POST 重新提交相同的 JSON，对已有的作业配置进行更新。如果你使用的是 cURL，只需要在上面的命令行上添加 -X PUT 的参数即可完成。

尽管这些独立的基于计划的作业已经展示了 Chronos 是怎样为 Mesos 提供一个

更全面的 Cron 系统的，但当你开始使用复杂的作业时，你才能看到 Chronos 的实力。

8.4 使用复杂的作业来工作

前面的小节涵盖了基于计划运行的简单且独立的作业，它们跟我们熟悉的 Cron 守护进程非常相似。Chronos 也有能力创建基于依赖的作业，该作业只有父作业成功完成时才会运行。尽管这种用例很多，但比较突出的是和数据处理及分析相关的 ETL 作业。

8.4.1 组合基于计划和基于依赖的作业

ETL 作业的核心工作如下：

- 从一个或者多个数据源抽取数据
- 转换数据并以更适合分析的格式储存
- 在目标存储中载入数据

在 Chronos 里，基于依赖的作业没有 schedule 字段，而是指定一个或者多个父作业（使用 parents 字段）完成后才能运行该作业。图 8.4 展示了一个 ETL 作业在 Chronos 中是怎样运行的。注意 schedule 字段和 paraents 字段的不同。

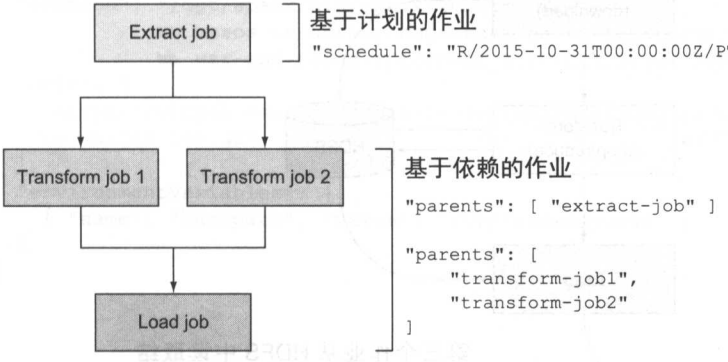


图 8.4 复杂的 Chronos 作业只有当所有的父作业完成时才会运行

在图 8.4 中，extract-job 必须成功完成后，transform.job1 和 transform.job2 才能开始。通过在作业间建立依赖关系，你能够开发更多复杂的计划作业。结合外部的数据存储，Chronos 在 Mesos 集群上为你提供可靠的计划作业的方式。

让我们看一下这个示例：调度 ETL 作业来对列夫·托尔斯泰的《战争与和平》一书的单词进行统计。

示例：获取《战争与和平》一书中频率最高的 20 个单词

为了展示如何在一个 ETL 作业的环境中结合基于计划和基于依赖的作业，我举了一个 Chronos 作业的例子，这个 Chronos 作业是由一个单独的基于计划的作业和两个基于依赖的作业组成的。这个例子也使用了第 2 章中简要提到的 HDFS 和 Sparks。你可以在本书的补充资料中找到这个例子的源代码。

> **提示：**为了使用 Chronos 运行 Spark 作业，我已经在 Mesos 集群上安装了 Cloudera 的 CDH 5.3 Hadoop 发行版本和 Apache Spark。相关的安装指引可以访问：http://www.cloudera.com/content/www/en-us/documentation/enterprise/5-3-x/cloudera-homepage.html 和 http://spark.apache.org/downloads.html。

为了让这个例子更形象，请看图 8.5。

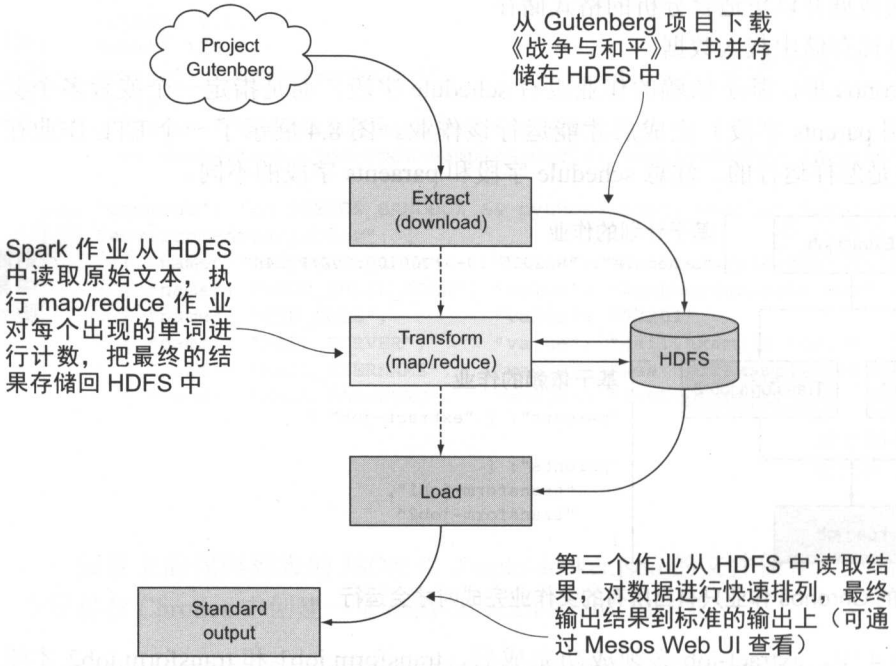


图 8.5《战争与和平》一书 ETL 作业的三个 Chronos 作业的可视化

尽管这看起来可能是个复杂的任务，但 Chronos 允许你创建更小的独立作业，然后通过创建依赖关系把它们链接在一起。在列表 8.3 中，第一个作业是从 Gutenberg 项目中下载《战争与和平》一书，然后保存结果文本到 HDFS 中。

清单 8.3 Chronos 作业下载《战争与和平》

```
{
  "name": "download-war-and-peace",
  "description": "Download the text of War and Peace.",
  "schedule": "R1//PT1M",
  "command": "hadoop fs -mkdir -p ${basepath} && hadoop fs -put -f
      ➥ pg2600.txt ${basepath}/warandpeace.txt",
  "uris": [ "http://www.gutenberg.org/cache/epub/2600/pg2600.txt" ],
  "environmentVariables": [
    { "name": "basepath", "value": "/tmp/warandpeace" }
  ]
}
```

按时间表运行一次这个作业

在 ETL 作业的下一步，你需要执行一个 Spark 的 map/reduce 作业来处理文本并且计算每个单词出现的次数。在下面的列表中，你创建第二个 Chronos 任务，这次在 parents 字段中指定上个作业（download-war-and-peace）

清单 8.4 运行一个 Spark 任务对《战争与和平》单词进行计数

```
{
  "name": "war-and-peace-wordcount-spark-job",
  "description": "Use Spark to count all the words in War and Peace",
  "parents": ["download-war-and-peace"],
  "command": "/opt/spark/bin/spark-submit
      ➥ mesos-in-action-code-samples-master/wordcount-example/
      ➥ war-and-peace-wordcount_2.10-0.1.0.jar ${basepath}",
  "uris": [
    "https://github.com/rji/mesos-in-action-code-samples/archive/
    ➥ master.tar.gz"
  ],
  "environmentVariables": [
    { "name": "basepath", "value": "/tmp/warandpeace" }
  ]
}
```

当 download-war-and-peace 顺利完成时运行这个作业

上面代码列表的 Spark 作业从 HDFS 读取文本，然后存储文本中单词的计数。为了在 Mesos UI 中显示输出，让我们读取列表 8.5 中 Spark 作业的结果，排序并拿到开始的 20 行。如在列表 8.4 中一样，可以通过指定 parents 字段来和上一个作业建立依赖关系。

清单 8.5 载入结果并且输出最多的 20 个单词

```
{
  "name": "load-war-and-peace-word-counts",
  "description": "Read the output from HDFS and send it to stdout",
  "parents": ["war-and-peace-wordcount-spark-job"],
  "command": "hadoop fs -cat ${basepath}/result/part-* |
              ➥ sort -t, -rnk2 | head -20",
  "environmentVariables": [
    { "name": "basepath", "value": "/tmp/warandpeace" }
  ]
}
```

当 war-and-peace-wordcount-spark-job 顺利完成时运行这个作业

就是这样！这三个作业会运行需要 ETL 管道来处理《战争与和平》的文本并且输出频率最高的 20 个单词。为了尽快地展示这个过程，在本书 GitHub 的版本库里有一个部署这三个作业的小脚本。如果你的 Mesos 集群上已经安装好 Spark 和 HDFS，可以运行如下命令：

```
$ complex-etl-job/create-jobs.sh http://chronos.example.com:4400
```

该脚本会创建如清单 8.3~ 清单 8.5 所定义的三个作业。假设一切如预料的那样实施，你可以浏览 Mesos 的 Web 接口，查看 `load-war-and-peace-word-counts` 作业的输出。它会显示书中出现频率最高的 20 个单词。

8.4.2 形象化作业的依赖关系

Chronos 有一个非常有用的特性是可以形象化大量作业之间的依赖关系。取代了手动地追踪复杂作业的依赖关系，你可以使用 Chronos 的 Web 接口查看动态生成的图像。图 8.6 展示了上一节中 ETL 管道创建的作业依赖关系。

依赖图另外一个有用的方面是它会突出显示哪个作业是成功的，哪个是失败的。这样你能够在越来越复杂的作业中快速地发现并且解决失败异常。

提示：相较于渲染图，如果你宁愿检查 DOT 文件的内容，那么你可以访问 /scheduler/graph/dot 这个 API 路径。

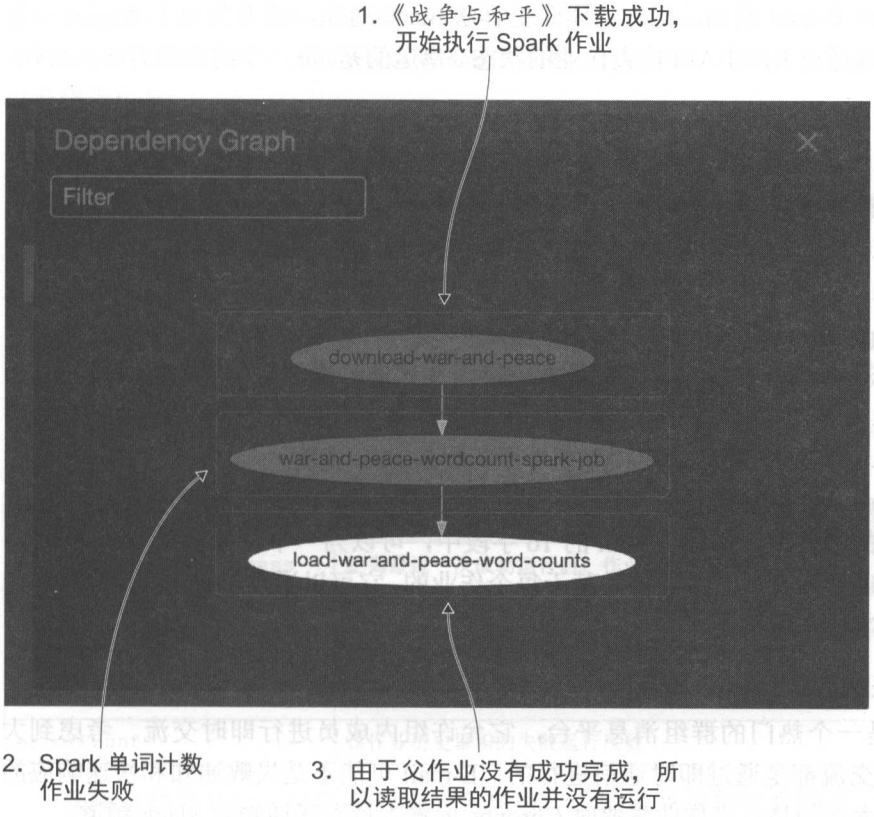


图8.6 Chrons的作业依赖图包含了作业的父子关系。第一个作业的成功触发了第二个作业的执行，但第二个作业的失败使得第三个作业从未执行过

8.5 监控Chronos作业的输出和状态

某个事情“是否”失败了不是问题，而“何时”才是问题。即便是设计得最好的系统都会遭受某种错误或者最终服务会降级。关键是要确保系统失败或者降级得温和一些并且在发生时能把相关信息通知到系统的所有者。

本节涵盖了如何启用 Chrons 的作业失败的邮件和 Slack（热门的群组消息服务）通知。本节也涵盖了如何监控作业来判断它们健康与否，以及观察作业的输出做进一步的调试。

8.5.1 作业失败事件的通知和监控

取决于你的组织、团队以及接收报警和通知时更喜欢的媒介，在作业失败时，

你可能更喜欢 E-mail 通知，但你的团队更喜欢 Slack 通知，或者类似于 Nagios 的监控系统定时地通过 REST API 检查作业的状态。幸运的是，每一个需求都有解决方法。

作业失败时的 E-mail 通知

要启用 E-mail 通知，Chronos 首先必须配置连接到邮件服务器。尽管我们在前面的安装和配置部分已经介绍过了，你还是要在 /etc/chronos/conf 里配置如下选项：

- `mail_from`——发送通知的发件人邮件地址，例如 chronos@example.com；
- `mail_server`——需要连接的邮件服务器，指定服务器和端口（例如 mail.example.com:25）；
- `mail_user`——可选，当邮件服务器需要认证时提供的用户名；
- `mail_password`——可选，当邮件服务器需要认证时提供的密码；
- `mail_ssl`——可选，启用到邮件服务器的 SSL 连接。

在 Chronos 实例配置好到邮件服务器的连接后，每个作业都可以指定一个或者多个所有者的邮件地址。在 E-mail 的 To 字段中，可以为一个作业输入多个所有者，只要确保用逗号隔开。这个选项是基于每个作业的，它可以通过 Web 接口和清单 8.2 中的 owner 字段来配置。

作业失败时的 Slack 通知

Slack 是一个热门的群组消息平台，它允许组内成员进行即时交流。考虑到大部分的组内交流都是通过即时通信进行的，E-mail 可能不是失败通知和快速响应的最佳选项。幸运的是，当作业失败时 Chronos 内置支持发送通知到 Slack 频道。

注意：如先前提到的，Chrons 中的 Slack 支持要求你使用 --slack_url 的配置选项来提供 Slack 的 Webhook URL。这点可以通过在每个 Chronos 实例上创建 /etc/chronos/conf/slack_url 文件来实现。

在本书编著时，Chronos 里的 Slack 实现有个小缺陷：由于作为 Chronos 实例的配置选项而非作业，所有的通知必须发送到同一个频道。在某些场景下，发送通知到 #general 或者 #sysops 都是可以的。图 8.7 展示了发送到 #general 频道的 Slack 通知示例。

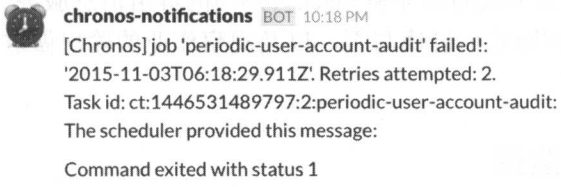


图 8.7 在 Slack 频道里的 Chronos 通知示例

如你在截图中所看到的，有一个作业失败了，其 Slack 通知也会发送出去。这个通知包含作业的名字、时间戳、重试过的次数，任务 ID（用于在 Mesos UI 中调试）以及错误信息。

通过 REST API 监控作业的状态

Chronos 提供 /scheduler/jobs 的 API 路径，用于获取 Chronos 内所有的作业信息、配置以及作业的成功与失败情况。通过简单的一个 HTTP 请求来遍历 JSON 数组里的项目。你能够检查所有作业的状态。

正如你会为监控系统写一个脚本去检查 Cron 作业的状态一样，你也可以对 Chronos 作业做一样的事情。在某个给定作业的 JSON 哈希中，特别需要注意的字段如表 8.3 所示：

表 8.3 监控用途的重要的可用作业指标

字　段	描　述
errorsSinceLastSuccess	距离最近一次成功运行后，作业运行失败的次数
lastSuccess	最近一次成功运行的日期和时间（ISO 8601 格式）
lastError	最近一次失败运行的日期和时间（ISO 8601 格式）
successCount	该作业历史累积的成功运行次数
errorCount	该作业历史累积的失败运行次数

获知这些信息后，可以根据你的选择写一个监控系统的检查脚本。下面是一些可以想到的报警用例：

- 作业错误状态报警：如果上个作业执行成功的话，errorsSinceLastSuccess 为 0。如果为正，则说明当前的作业处于错误状态。
- 作业运行状态不满足 SLA 报警：lastSuccess 是指作业最近一次成功的时间戳，当时间间隔已经超过 SLA 规定，例如某个每小时运行的关键的业务数据处理作业，在过去的三个小时都没有成功的话，lastSuccess 参数能够用来升级报警给工程师或者产品经理。
- 特定作业的失败率报警：通过公式 failure-pct= errorCount/(errorCount+successCount)，你可以创建一个检查条件来判断一个作业成功完成的频率。

8.5.2 通过 Mesos 观察作业的标准输出和标准错误

由于 Chronos 作业是以 Mesos 任务的方式运行的，所以观察这些作业输出的最

简单方法就是使用第 5 章提到的 Mesos Web 接口。再复习一下，在任务沙盒中有两个文件会被自动创建：stdout 和 stderr，如图 8.8 所示。这些文件分别捕获了任务的标准输出和标准错误。

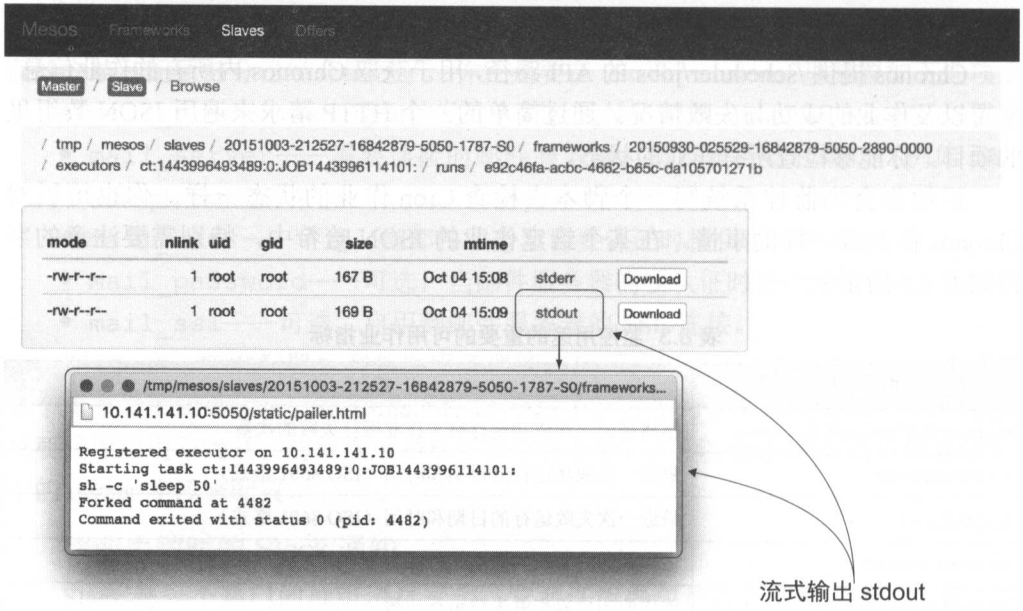


图 8.8 在 Mesos Web 接口上观察 Chronos 作业的输出

点击一个任务沙盒内的 stdout 或者 stderr 连接会打开一个新窗口，它会实时地把作业的日志以流的方式更新到 Web 浏览器里。因为不需要再登录到某台机器上，这让观察一个长时间运行的 Chronos 作业变得简单或者无须登录到机器中就可判断作业失败的原因。

8.6 小结

在本章你学习了在 Chronos 上部署计划作业。你探索了几个主题，例如安装和配置、作业和作业依赖，以及监控。下面几点是需要记住的：

- Chronos 作业的计划以 / 为分隔符，分成三部分。作业的重复次数，作业的开始时间（ISO 8601 格式）以及运行的间隔（ISO 8601 标准的时长）。
- 按计划运行且没有任何父子依赖的作业称之为简单的作业。而复杂的作业指的是拥有父作业，并且一个或多个子作业必须在父作业成功完成后才能运行。
- 当使用 Docker 镜像时，Mesos 的任务沙盒会自动映射到容器里。这使得你可以在容器启动前，使用 Mesos 提取器把需要的文件下载到沙盒中。

- 当创建依赖父作业的作业时，`schedule` 字段是被省略的，相反，你必须使用 `parents` 字段。
- 在你能以多种方式来监控作业、发送失败通知或者观察作业输出：可以通过 E-mail 和 Slack 发送通知；作业的状态，包括成功和失败的次数与实践都能通过 REST API 获得；而观察作业的输出只需要检查任务沙盒内的 stdout 文件和 stderr 文件。

尽管本章使用了最常用的参数提供了几个深入的示例，但 Chronos 还有很多其他本书未能涵盖的特性，而且在每次发布后，也会增加越来越多的特性。关于 Chronos 的最新文档，请访问 https://mesos.github.io/chronos。

第 9 章涵盖了热门的 Apache Aurora 项目，这个单独的 framework 能让你在 Mesos 上部署应用和计划作业。

9 使用Aurora部署应用和管理计划任务

本章内容

- 构建，安装和配置 Aurora
- 部署应用和 Docker 容器
- 创建计划任务

在前面的两章，你已经学习了如何使用 Marathon 和 Chronos 这两种流行的 Mesos framework 进行应用部署和执行计划任务。本章中，我将介绍 Apache Auora，一个由 Twitter 开发出来用于简化应用部署操作和 Cron 作业的 Mesos framework。

尽管 Aurora 的配置和部署比 Marathon 或者 Chronos 要复杂，但本书介绍 Aurora，是为了在你部署自己的 Mesos 集群时，多一个 framework 上的选择。举个例子，Marathon 和 Chronos 都提供了易于使用的基于 JSON 的 REST API，但是它们不提供用户访问权限控制。另一方面，Aurora 提供多用户的权限控制，但同时也增加了其基于 Python 开发语言的复杂性。当然，你需要评估开发或预生产环境的各个环节来决定哪个最适合你的团队和组织。尽管 Aurora 有点复杂，但是功能强大。

在运行 Aurora 之前，你应该熟悉软件的编译，编写自定义的服务脚本和（可选）构建自定义的应用包。因为在第 7 章和第 8 章已经介绍了如何运行应用和定时作业，

本章我不一定会重复所有的内容，而是采用你已经熟知的例子来介绍 Aurora 的不同之处。如果你还没有阅读第 7 章和第 8 章而又对 Mesos 如何处理应用部署和计划任务感兴趣，我建议你先阅读这些。那么，让我们开始吧！

9.1 Aurora简介

2010 年，Aurora 开始作为 Twitter 内的一个项目，而当时，Twiter 公司刚通过使用 Mesos 来缩放其基础架构，从而使“失败之鲸”成为往事。作为最早的 Mesos framework 之一，Aurora 通过提供自助服务应用和作业管理，为 Twitter 庞大的开发团队提供了便捷的操作。

Twitter 开源了这项成果，Aurora 和 Mesos 一样，成为了 Apache 软件基金会的顶级项目。可以肯定地说，Aurora 已经在大规模生产环境中得到验证，并每天支持着数以百计的软件工程师部署应用和 Cron 作业。

注意：本章使用 Aurora 0.9.0 版。

Aurora 为满足对多用户认证的支持，所以它不像（某些情况下）其他的 Mesos framework 一样功能齐全。但你可以放心，Aurora 中的功能都是稳定鲁棒的。让我们通过表 9.1 来对比一下 Aurora 和之前讨论过的其他 framework（Marthon 和 Chronos）。

表 9.1 Aurora、Marathon 和 Chronos 的功能比较

	Aurora	Marathon	Chronos
DSL 配置	是	否	否
部署常驻应用	是	是	否
部署计划（定时）作业	是	否	是
Docker 支持	实验 / 不完整	是	是
HA 集群部署	是	是	是
安装包支持	否	是	是
REST API	否	是	是
支持用户认证和资源分配	是	否	否

尽管 framework 通常被称为 Aurora，但它是由多个组件构成的：

- 调度器；
- 客户端（两个，一个用户，一个管理员）；
- 执行器；

- 观察者。

上面所有组件共同构建了一个在 Mesos 集群上运行应用和计划作业的平台。调度器作为 Aurora 集群的主要接口提供服务，命令行客户端为用户提供了创建、更新和删除作业的方法。执行器和观察者提供运行和监控任务的一致的执行环境，管理员客户端则允许进行集群管理。下面让我们详细地了解一下各个组件。

9.1.1 Aurora 调度器

和其他 Mesos 调度器一样，Aurora 的调度器负责向 Mesos master 进行注册，接受或减少资源提供，在集群上启动任务。和你在本书已经学到的其他 framework 一样，调度器是负责给用户提供整个 Aurora 服务的组件。当用户需要部署新的服务或者计划作业时，调度器控制进程被分派到集群的哪一台机器上。

调度器本身是由 Java 编写的，在构建过程中，会将它所有的依赖包都打包到一个单独的 JAR 中（除了 libmesos 和 JVM 本身，这两个必须已经安装在系统上）。与 Mesos 和其他 Mesosframework 一样，Aurora 使用 ZooKeeper 来进行复制检测、leader 选举和提供服务发现信息。通过使用类似 ZooKeeper 的分布式数据库，Aurora 可以被部署成高可用模式。

大部分的 Aurora 操作都是通过调度器发起或控制的。连接调度器的命令行客户端被用于管理用户作业和集群本身。借助 Apache Shiro 安全 framework，调度器还可以支持多用户，关于 Apache Shiro，我将在本章的稍后部分进行介绍。最后，调度器还提供了一个 Web 接口，用于查看作业配置、各个应用配置和部署状态。

9.1.2 Thermos 执行器和观察者

之前你已经了解到，Mesos 是通过执行器将一个任务运行在集群的某一台 slave 上的。`CommandExecutor` 是 Mesos 的一个内置执行器，通过 shell(/bin/sh)运行命令。同时，Mesos 也提供了一个 API 来开发自定义执行器并实现任务的启动、监控和删除。Thermos 是 Aurora 使用的一个基于 Python 开发的自定义执行器，它为调度器启动任务提供一个一致的执行环境。

提示：第 10 章将会更详细地介绍自定义执行器。

Thermos 由两个部分组成：

- 执行器——一个 Mesos 的执行器，可以运行和管理 Aurora 的任务和健康检查。
- 观察者——一个独立服务（每个 Mesos slave 上一个实例），负责监控 Thermos 启动的任务状态。

当 Aurora 的调度器从一台 Mesos slave 上接收到可供任务执行的资源提供时，它首先启动执行器。一旦执行器注册到 Mesos slave 上，它就可以开始启动用户请求

的进程，不论是启动一个应用实例还是运行一个计划作业。

Aurora 的观察者的独特之处在于，它会对 Aurora 集群中的每一个 Mesos slave 上的执行器进行监控，并提供 Web 接口。按照这种方式，用户可以与每个集群节点的监控服务进行交互，而不需要访问 Mesos 基础架构本身。可能你还记得在第 7 章，Marathon 展示了一些基本的任务信息，但是更加细节的内容，你需要切回到 Mesos 的 Web 接口上（至少在撰写本书时）。Aurora 调度器和 Thermos 观察者的 Web 接口的紧密集成为用户提供了一致的用户体验，也降低了在复杂部署过程中的成本。

9.1.3 Aurora 的用户和管理员客户端

Aurora 提供了两个独立的基于 Python 的命令行客户端，我称之为*用户客户端*和*管理员客户端*。根据名字理解，用户客户端是给应用开发人员进行服务和计划作业部署的。而管理员客户端是给系统管理员进行集群管理和用户管理等操作的。

本章的稍后，你会了解更多关于如何使用这两个客户端的内容。现在，我将对这两个客户端所提供的功能做一个简要的概述。

用户客户端

用户客户端通常被部署到用户和工程团队，使得他们可以在同一个集群中管理自己的服务和计划作业。几个值得注意的功能如下。

- 作业——创建，删除，重启，罗列服务和作业。
- `Cron`——创建，修改，移除和手动启动 Cron 任务。
- 更新——开始，中止，暂停，重新开始一个服务或者作业的滚动更新。

客户端内置了大量的功能，并还在不断地扩展。获取命令的详细列表或使用说明和帮助，请执行 aurora -h 或者参考官方文档：http://aurora.apache.org/documentation/latest/client-commands。

管理员客户端

管理员客户端是定制给系统管理员进行 Aurora 集群维护的。你可以通过笔记本或服务器访问指定的 Aurora 集群进行操作。它包含以下几个功能。

- *备份和恢复*——对调度器状态进行备份和从备份文件进行恢复。
- *维护*——将主机变更为（或移除出）维护状态，包括将作业从一台或一组特定主机中迁移出来。
- *配额*——为特定用户创建和修改生产配额。
- *SLAs*——探测特定的主机，判断主机是否到达预设值而决定是否离线。

和用户客户端一样，管理员客户端也内置了大量功能，我就不在这里重复了。获取子命令完整列表，请运行 `aurora-admin -h`。我将在本章后面介绍一些主要

的功能，特别是主机维护和用户管理。

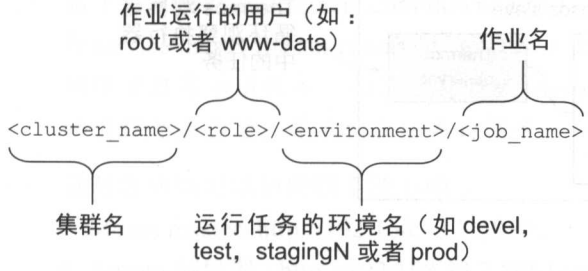
9.1.4 Aurora DSL（Domain-Specific Language，特定领域语言）

不像 Marathon 和 Chronos，Aurora 使用基于 Python 的 DSL 来进行作业配置，而不是基于 JSON 的 API。每个配置文件都由部署应用或作业的团队维护管理，并以 .aurora 作为后缀。

Aurora 的语言比较简单，值得了解：

- `Process`—— 在系统中执行的命令。一个 Task 由一个或多个 Process 组成。
- `Task`—— 一个 Task 包含一个或多个 Process，即有两种形式：`SimpleTask` 和 `SequentialTask`。
- `Job`——Tasks 的集合，可以运行在一个 Mesos slave 上

这些配置文件由 Aurora 客户端处理并部署到集群中。每一个作业都有唯一的标示，称之为作业键值，键值的格式如下：



一个 Aurora 的作业在配置文件中以作业键值的形式出现，包含一个或者多个 `Process`、`Task` 和 `Job`。在本章后面我会介绍一些由 Aurora DSL 编写的作业的例子。完整的配置参考和使用 Aurora DSL 配置文件的最佳实践请参考如下的官方文档。

- DSL 的配置教程：http://aurora.apache.org/documentation/latest/configuration-tutorial。
- 整个模式详细的配置指引：http://aurora.apache.org/documentation/latest/configuration-reference。

现在，你已经熟悉了 Aurora 的组件——调度器、执行器、观察者、两个命令行客户端和作业配置使用的 DSL，让我们学习如何构建、安装和配置 Aurora 到你的 Mesos 集群。

9.2 部署Aurora

Aurora 与 Mesos 和其他 Mesos framework 一样，有多种部署方式。它可以作为

一个单独的实例安装（用于开发），或具有高可用性（用于生产），或者你只是想在开发环境中测试一下，那么可以在你电脑上使用从 aurora.apache.org 下载的 Vagrant 文件来搭建一个单节点的 Aurora 集群。

在继续之前，我们需要先了解 Aurora 的各个组件是如何交互的，如图 9.1 所示。

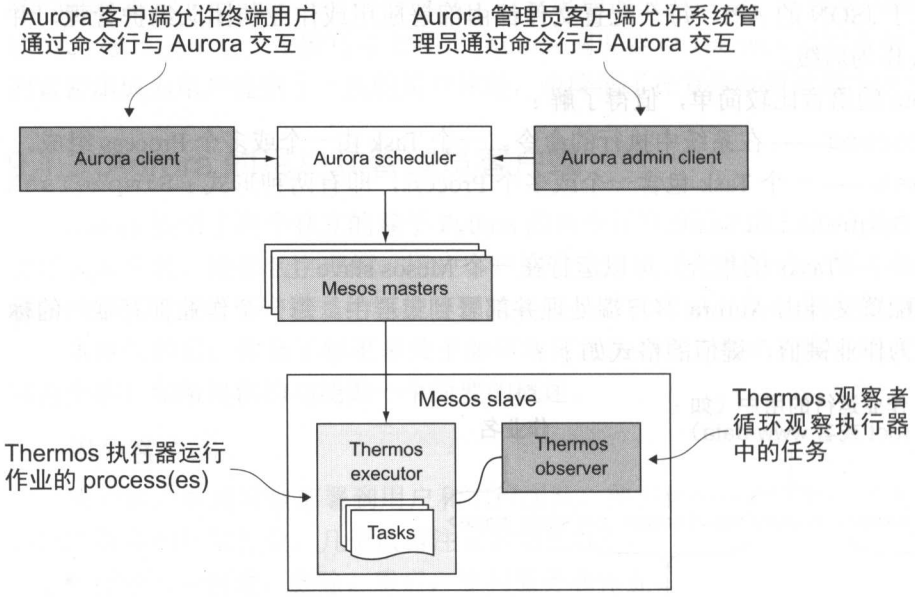


图 9.1 Apache Aurora 组件图

和其他 Mesos framework 的调度器一样，Aurora 的调度器向 Mesos master 注册，并在 slave 上启动任务。Aurora 命令行客户端负责部署应用，创建 Cron 任务和管理集群。当调度器在一个集群上工作时，它从主 Mesos master 上获取资源邀约，运行一个或者多个任务。

本节将讨论如何在开发环境（Vagrant）中启动和运行 Aurora，而如何将它部署到生产 Mesos 集群，我们已经在第 3 章中介绍过。

9.2.1 在开发环境尝试 Aurora

学习 Aurora 最简单的方式就是在开发中直接使用带有项目 Git 库的 Vageant 环境。如果你不熟悉 Vagrant，那么这是一个开源的工具，它提供了一种简单的方式与他人分享开发环境。你可以在 www.vagrantup.com 找到更多的信息，包括下载。

下载的 Aurora 环境的 Vagrant 文件包含了 Aurora、Mesos master、Mesos slave，同处在一个虚拟机上。你可以在你的笔记本电脑或台式机上尝试 Aurora，部署一些应用或 Cron 作业。如果你打算按照生产的方式来安装 Aurora 的话，请跳到 9.2.2 节学习。

启动 Vagrant 环境

Vagrant 的环境通常用于开发目的启动 Aurora，你可以通过运行以下的命令，从 Apache Software 基金会的 Git 库克隆一份 Aurora。

```
$ git clone git://git.apache.org/aurora.git
$ cd aurora
$ git checkout rel/0.9.0
```

当克隆完成后，检查一下 Aurora 版本是否和本章介绍的一致，然后通过下面的命令启动 Vagrant 虚拟机：

```
$ vagrant up
```

注意：Vagrantfile 包含 Aurora 0.9.0 和 Mesos 0.22.0，和当前版本的 Docker 有一些同样的问题。当 Vagrant 的虚拟机启动后，你需要先将 Mesos 升级到 0.22.2 版本，然后添加 Mesosphere 的扩展包（第 3 章中提到的或者在网站下载 https://mesosphere.com/downloads）。升级 Mesos 之后，需要通过运行 service mesos-master restart 和 `service mesos-slave restart` 命令，确保重启每一项服务。如果没有升级 Mesos，会出现 `mesos-slave` 的守护进程无法启动，并不响应任务的调度。

这时你应该可以访问以下的 URL。

- Mesos master: http://192.168.33.7:5050。
- Aurora 调度器 : http://192.168.33.7:8081/scheduler。
- Thermos 观察者 : http://192.168.33.7:1338。

9.2.2 构建和安装 Aurora

和 Mesos 一样，你需要在一些机器上部署 Aurora 的调度器，在生产工作负载下，三台或五台 Aurora 调度器实例就应该是足够的。为保持部署得简单，你可以将 Aurora 部署到和 Mesos master 相同的机器上。Aurora 项目在撰写本书时还没有 RHEL 和 Ubuntu 的安装包支持，所以我选择从 aurora.apache.org 上下载源码进行 Aurora 的创建。

提示：尽管在撰写本书时没有可用的 Aurora 安装包，但他们正在努力构建和发布官方包中。你可以在 https://git-wip-us.apache.org/repos/asf?p=aurora-packaging.git;a=tree 上获取最新的代码。但是现在，你需要为你的服务管理工具或进程管理工具（systemd，upstart，supervisord 等）编写自己的服务脚本，或者尝试根据自身需要修改代码包中的脚本。

为方便构建 Aurora 的各个组件，并保持构建命令尽可能简单，我已经在本书附带的 GitHub 资源包第 9 章中提供了一份脚本。这个脚本可以在 RHEL/CentOS 7 和 Ubuntu 14.04 中构建 Aurora 组件。

构建 Aurora

要构建构成 Aurora 的所有组件，首先要通过运行以下命令，克隆一份本书的 Git 库：

```
$ git clone https://github.com/rji/mesos-in-action-code-samples
$ cd chapter09
```

然后，使用 aurora-build.sh 这个脚本编译所有的组件。脚本在执行中还会安装 Aurora 所依赖的任何必备包。

```
$ ./aurora-build.sh all
```

构建脚本耗时几分钟，当它完成时，所有构建好的组件都会在 apache-aurora-0.9.0/dist 目录下。接下来的部分，我会对单个组件的安装进行说明。

安装 Aurora 调度器

正如我之前提到的，假设你是在每一台 Mesos master 服务器上安装 Aurora 调度器实例，而部署 Aurora 调度器的 zip 文件放在之前构建生成的目录中 apache-aurora-0.9.0/dist/distributions/aurora-scheduler-0.9.0.zip，请将它复制到所有的 Mesos master 机器上：

```
$ scp apache-aurora-0.9.0/dist/distributions/aurora-scheduler-0.9.0.zip
➥ user@mesos-master-1.example.com:
```

接下来，将 home 目录下的包解压到指定目录，本书中，我们将它安装到 /usr/local 中，并且创建一个 /usr/local/aurora-scheduler 的软链接。这可以使你以后在进行 Aurora 版本升级时，只须将新版本放到 /usr/local 目录中，并更新软链接的目标即可。

```
user@mesos-master $ sudo unzip aurora-scheduler-0.9.0.zip -d /usr/local
user@mesos-master $ sudo ln -fns /usr/local/aurora-scheduler-0.9.0
➥ /usr/local/aurora-scheduler
```

随着 Aurora 调度器代码在各个 Master 中就位，让我们继续安装其他的组件。并且我将在 9.2.3 节中讲解各个组件的配置。

安装 Thermos 执行器和观察者

安装 Thermos 执行器和观察者，你需要将几个执行文件复制到每一台 Mesos 集群的机器上。对于大型安装，借助于像 Puppet、Ansible 等的配置管理工具将方便很多。

将你本地机器上apache-aurora-0.9.0/dist目录下的thermos_executor.pex和thermos_observer.pex复制到Mesos slaves的每一个Mesos slave上，通过运行以下命令：

```
$ scp -p apache-aurora-0.9.0/dist/thermos_{executor,observer}.pex
➥ user@mesos-slave-N.example.com:
```

接下来，和处理调度器一样，将它们移到磁盘上合适的位置，并创建指向当前版本的软链接。这样你可以通过更新软链接的目标实现执行器和观察者的代码升级。

```
user@mesos-slave $ sudo mkdir /usr/local/aurora-executor-0.9.0
user@mesos-slave $ sudo mv thermos_{executor,observer}.pex
➥ /usr/local/aurora-executor-0.9.0/
user@mesos-slave $ sudo ln -fns /usr/local/aurora-executor-0.9.0
➥ /usr/local/aurora-executor
```

注意：Aurora执行器代码在集群中的每一台slave上的路径必须相同。在下一节配置Aurora调度器的时候需要使用这个路径。

将Aurora的代码复制到Mesos的master和slave上后，你差不多就可以开始部署应用和创建Cron作业了。但在此之前，你还需要先构建和安装命令行客户端。

安装Aurora客户端

Aurora提供了一个独立的命令行客户端，可以为大大小小的工程组织提供从开发机器上直接管理作业和服务的自助式操作方式。当运行完aurora-build.sh脚本构建所有的组件后，你可以在apache-aurora-0.9.0/dist目录下找到一个名叫aurora.pex的文件，将它移动到客户端机器的合适目录下，通过运行以下命令：

```
$ sudo mv apache-aurora-0.9.0/dist/aurora.pex
➥ /usr/local/aurora-client-0.9.0.pex
$ sudo ln -fns /usr/local/aurora-client-0.9.0.pex /usr/local/bin/aurora
```

如果一切正常，你应该可以执行`aurora --version`命令，并返回0.9.0。如果不是，请确保/usr/local/bin配置到你的$PATH下面。

构建和安装Aurora管理员客户端

Aurora管理员客户端是Aurora的第二个命令行工具，是专为系统管理员设计并提供了集群运维和集群管理的相关工具。执行下面的语句，将`aurora-admin`命令行工具安装到/usr/local/bin/aurora-admin：

```
$ sudo mv apache-aurora-0.9.0/dist/aurora_admin.pex
➥ /usr/local/aurora-admin-0.9.0.pex
$ sudo ln -fns /usr/local/aurora-admin-0.9.0.pex
➥ /usr/local/bin/aurora-admin
```

现在，你已经构建和安装了 Aurora 的所有组件，让我们开始配置它们。

9.2.3 配置 Aurora

每个 Aurora 的组件，特别是调度器，都是可以根据你特定的期望、需求和环境进行深度的配置的。本节讲解如何对 Aurora 进行配置，包含必须（和一些推荐）的配置项。

调度器的配置

Aurora 的调度器负责和 Mesos 对接、启动任务和供用户提交作业，可以说，大部分的操作需要依赖它进行启动和运行。让我们先从配置 Aurora 的调度器并保证服务的启动和运行来开始配置 Aurora。

注意： *Aurora 正常情况下是由外部进程监控工具启动的，如 systemd、upstart 或者 supervisord，在你构建和部署 Aurora 时需要时刻牢记。*

表 9.2 包含了必需的配置项和在 Mesos 集群上部署 Aurora 调度器时可能需要的一些重要的配置项。希望获取完整的配置项清单，你可以运行以下命令：

```
$ /usr/local/aurora-scheduler/bin/aurora-scheduler -help
```

表 9.2 Aurora 调度器必需（重要）配置项

配置项	说 明
-cluster_name	任意名，用于识别 Aurora 集群
-mesos_master_address	检测和连接到 Mesos master 的 ZooKeeper URL，如：zk://zk1:2181,zk2:2181,zk3:2181/mesos
-serverset_path	ZooKeeper SeverSet 路径，用于注册，如 /aurora/scheduler
-zk_endpoints	Aurora 使用的 ZooKeeper 集合服务器清单，如 zk1:2181,zk2:2181,zk3:2181
-native_log_quorum_size	Aurora 调度器推选 leader 的个数，如果是部署在 Mesos master 上的，此值为 /etc/mesosmaster/quorum
-native_log_file_path	Aurora 使用的 Mesos 复制日志持久化文件存放的位置
-backup_dir	Aurora 存储备份的目录
-thermos_executor_path	Slave 上 Toermos 执行器所在的路径，集群所有 slave 上的执行器路径必须一致
-thermos_executor_flags	传递给 Thermos 执行器的额外选项
-allowed_container_types	Aurora 支持的容器，像我们的集群，可以支持 DOCKER，MESOS

续表

配置项	说明
-framework_authentication_file	Mesos 认证使用的文件（见第 6 章）
-zk_digest_credentials	登录 ZooKeeper 的身份验证文件，格式：用户名：密码

每一个调度器实例为了维护它们的状态，Aurora 使用一个 mesos 的复制日志实例。因此，你需要在每一台 Mesos master 上初始化一个新的 Aurora 专属的复制日志。运行下面的命令将在 /var/db/aurora 路径初始化一个日志文件：

```
$ sudo mkdir -p /var/db && sudo mesos-log initialize --path=/var/db/aurora
```

注意： Aurora 的 Mesos 复制日志路径是完全任意的，但是它必须与 -native_log_file_path 传到 Aurora 的调度器中的值匹配。

因为 Aurora 的启动和配置比 Marthon 或者 Chronos（某些情况下，甚至是 Mesos）要复杂，所以我认为最好引入一个示例脚本来启动 Aurora 调度器服务。下面清单中的脚本还包含了给表 9.1 中各个配置项一些健全的赋值。

清单 9.1 Aurora 调度器启动脚本示例

```
#!/bin/bash

AURORA_SCHEDULER_HOME=/usr/local/aurora-scheduler
export LIBPROCESS_PORT=8083
export JAVA_OPTS="-Djava.library.path=/usr/lib -Xms2g -Xmx2g"

AURORA_OPTS=(
    -zk_endpoints=$(cut -d / -f 3 /etc/mesos/zk)
    -mesos_master_address=$(cat /etc/mesos/zk)
    -native_log_quorum_size=$(cat /etc/mesos-master/quorum)
    -cluster_name=aurora-cluster
    -http_port=8081
    -serverset_path=/aurora/scheduler
    -native_log_zk_group_path=/aurora/replicated-log
    -native_log_file_path=/var/db/aurora
    -backup_dir=/var/lib/aurora/backups
    -vlog=INFO
    -logtostderr
    -allowed_container_types=DOCKER,MESOS
    -thermos_executor_path=/usr/local/aurora-executor/thermos_executor.pex
    -thermos_executor_flags="--announcer-enable
    ➥ --announcer-ensemble $(cut -d / -f 3 /etc/mesos/zk)"
)

exec "${AURORA_SCHEDULER_HOME}/bin/aurora-scheduler" "${AURORA_OPTS[@]}"
```

Mesos 调度器驱动（libmesos）和 master 通信使用的端口

JVM 配置项；在这个示例中，我们指定了 heap 的大小和 libmesos 的位置

要注意的是，这个清单中的脚本只是一个如何配置服务的例子，根据你怎么配置 Mesos，或者你没有将 Aurora 部署在 Mesos master 守护进程所在的服务器上，在你的环境中这些都将不同。此外，你也可以像之前 Marathon 和 Chronos 的使用方式将脚本进行修改，也就是把所有的配置项都独立存放到一个文件中，服务脚本在执行时读取，作为参数传入。无论哪种方式，你都需要结合你的服务管理工具去运行调度器。

因为你有很多的部署项，且很多都是与个人偏好相关，所以让我们把 Aurora 调度器作为后台进程运行，这样你就可以继续在前台操作。如果你把清单 9.1 的脚本保存成 aurora-scheduler.sh，则可以按以下命令启动调度器：

```
$ chmod +x aurora-scheduler.sh
$ sudo ./aurora-scheduler.sh > /dev/null 2>&1 &
```

如果一切正常，你可以在 Mesos 的 Web 页面中看到 framework 已经以 TwitterScheduler 的名字注册了。请注意，为了简单，之前命令的所有日志输出都被重定向到了 /dev/null。---service manager of choice---- 当你把这个脚本用到你的服务管理工具中时，请确保日志重指向你系统的健全路径或使用日志集中的方案。

配置 Thermos 执行器和观察者

Thermos 执行器不需要任何的额外配置，可以直接使用。它只需要被部署到每台 Mesos slave 的相同路径。但与此同时，你可以使用很多配置项来微调执行器。因为这些选项用于微调而不是正常运行所必须的，我就不在这进行介绍了。获取完整配置参数的列表，请运行以下命令：

```
$ /usr/local/aurora-executor/thermos_executor.pex --long-help.
```

另一方面，Thermos 观察者是一个独立于执行器、运行在每一台 Mesos slave 机器上的服务。观察者提供正在运行的执行器和任务的信息，它需要少量的配置。表 9.3 列出了设置观察者的配置项。

表 9.3 Thermos 观察者配置项

配置项	说明
--root	查找 Thermos 执行器任务的根目录，默认为 /var/run/thermos
--mesos-root	Thermos 执行器沙箱所在的 Mesos 根目录，这里应该设置成 Mesos slave 的 --work_dir 一样的目录，默认为 /var/lib/mesos
--port	观察者的侦听端口，默认为 1338
--polling_interval_secs	Thermos 执行器任务状态的轮询尝试之间的时间间隔，默认为 5 秒

上面的表列出的是最重要的选项，但是和执行器一样，还有一些微调的选项我

没有在这里细讲。获取所有可用配置参数的完整清单，请运行以下命令：

```
$ /usr/local/aurora-executor/thermos_observer.pex --long-help
```

根据你的偏好和环境，需要使用一个外部进程管理工具来启动服务，或当服务意外中止时自动修复。而为了尽快将所有服务启动，可以运行下面的命令来启动Thermos观察者：

```
$ sudo /usr/local/aurora-executor/thermos_observer.pex --port=1338
➥ --log_to_disk=NONE --log_to_stderr=google:INFO > /dev/null 2>&1 &
```

现在，让我们来创建Aurora命令行客户端所需要的配置文件。

配置客户端

Aurora客户端使用JSON格式的配置文件，你可以定义一个或者多个的集群配置。默认情况下，客户端在/etc/aurora/或者~/.aurora/的路径下读取名为clusters.json的配置文件。文件的结构如下：

```
[
  {
    "auth_mechanism": "UNAUTHENTICATED",
    "name": "aurora-cluster",
    "scheduler_zk_path": "/aurora/scheduler",
    "slave_root": "/var/lib/mesos",
    "slave_run_directory": "latest",
    "zk": "mesos-master-1,mesos-master-2,mesos-master-3"
  }
]
```

通过在最顶层的数组下创建额外哈希表，可以在一个配置文件指定多个Aurora的集群，每个都有唯一的集群名。上面的例子已经足够让你启动并运行，但是完整的客户端配置请参看http://aurora.apache.org/documentation/latest/client-cluster-configuration。

注意：clusters.json配置文件中的ZooKeeper主机列表假定每个Zookeeper实例默认侦听2181端口，尽管有一个额外的zk_port配置项，但我故意没有使用，因为Aurora 0.9.0客户端有个Bug，zk_port只能作用于zk选项的最后一个主机，其他还是2181端口。详细信息可查看https://issues.apache.org/jira/browse/AURORA-1405。

到目前为止，所有的Aurora组件都已经构建、配置、部署到了Mesos集群和服务器，你可以浏览下面的URL来确保调度器和观察者服务是启动并运行的，并且Web页面是可以访问的。

- Aurora 调度器：http://mesos-master.example.com:8081/scheduler。
- Thermos 观察者：http://mesos-slave.example.com:1338。

如果一切正常，让我们开始在 Aurora 上部署一些实际的应用和计划作业。

9.3 部署应用

目前，越来越多的组织开始借助技术更加有效地经营他们的业务，不管是交付应用的增强性给用户（内部和外部），建立数据分析渠道来更好地了解用户行为，还是构建一个杀手级的应用，重要的是要有鲁棒的工具帮助你快速地部署更新。

越来越多的独立工程团队直接将应用的更新部署到生产环境（通常通过自动化系统的构建、测试和部署代码），以减少交付时间。因此，在应用部署过程中典型的没有权限这座“大山”被移除掉了，开发人员不再需要等待应用管理员或者系统管理员来部署他们的新代码。

Aurora 给工程团队提供了这种自助式的应用管理，工程师负责建立自己的 Aurora 配置文件并借助 Aurora 命令行工具以一种更适合指定应用的方式部署更新，Aurora 允许你定义应用的配置和需要运行的实例个数，并执行运行中应用的滚动更新，以减少服务不可用的时间。

使用 Aurora 时的服务发现

Aurora 的服务发现比你在第 7 章和第 8 章学到的方案要复杂得多，让我们花点时间来考虑一下，当使用 Aurora 部署服务时，有哪些方式可以让服务之间相互通信。

首先，Aurora 有一个内置的机制可以让执行器通告服务进入 ZooKeeper 中的 SeverSet，但是因为这个需要编写额外的代码和服务，我不会在本书中介绍。关于这个主题的更多信息，包括实现细节，可以查看：http://aurora.apache.org/documentation/latest/user-guide/#service-discovery。

其次，TellApart 已经写了一个名为 Aurproxy 的服务，其具有 Aurora 服务发现机制方面的知识。Aurproxy 可以通过在 Docker 容器中运行 Nginx 实现对 Aurora 上运行的应用实例流量的负载均衡。更多 Aurproxy 的信息，请查看 https://github.com/tellapart/aurproxy。

最后，正如我在第 7 章中提到的，Mesos-DNS 是一个基于 DNS 的 Mesos 服务发现机制，它可以通过 Mesos 运行中的任务的相关信息来创建 A 记录和 SRV 记录。因此，Mesos-DNS 自动地支持 Aurora，成为一个通用的、多 framework 的服务发现解决方案。更多 Mesos-DNS 的信息请查看：http://mesosphere.github.io/mesos-dns。

与 Marathon 和 Chronos 提供的基于 JSON 的 REST API 相比，Aurora 提供一个强大的 Python DSL，让你可以像编码一样创建和管理应用。因为这些定义可以使用 Python 编程语言的全部功能，你还可以在团队内外创建和重用模板。下面几个部分提供了如何借助 Aurora DSL 和命令行客户端来配置和部署应用。

> **提示**：如果你有兴趣学习 Twitter 是如何使用 Aurora 来部署应用的，可以观看 Bill Farner 在 2015 年 MesosCon 上的演讲“Generalizing Software Deployment”：www.youtube.com/watch?v=y1hi7K1lPkk。

9.3.1 部署一个简单的应用

在第 7 章中介绍 Marathon 时，我提到了本书的补充材料中有个名为 OutputEnv 的示例应用，这个简单的 Ruby Web 应用以格式化 Web 页面形式为当前应用实例（简单的 Mesos 任务）输出环境变量。

要在 Aurora 上部署这个应用，你首先需要使用 Aurora DSL 创建一个配置文件。在这个文件中，你需要定义角色、环境和你想要部署的作业，还有应用所运行的集群。下面的清单展示了如何使用 Aurora 部署这个应用。

清单 9.2 OutputEnv 应用的 Aurora 配置

```
tarball = 'https://github.com/rji/mesos-in-action-code-samples/
➥ archive/master.tar.gz'

download = Process(
    name='download', cmdline=' '.join(['curl -LO', tarball]))

extract = Process(name='extract', cmdline='tar zxf master.tar.gz')

run = Process(name='run', cmdline="""
    cd mesos-in-action-code-samples-master/output-env-app && \
    bundle install --retry 3 && \
    PORT={{thermos.ports[http]}} bundle exec ruby app.rb""")

task = SequentialTask(
    processes=[download, extract, run],
    resources=Resources(cpu=0.1, ram=128*MB, disk=1*GB))

jobs = [
    Service(
        cluster='aurora-cluster',
        role='www-data',
        environment='prod',
        name='outputenv',
        task=task,
        instances=3)
]
```

从 GitHub 下载本书的补充代码

解压 tar 包

运行命令启动程序，开放一个可用端口

为指定的集群、角色和环境创建一个新的服务

创建一个新的 SequentialTask 执行下载、解压和运行流程

配置文件中使用的集群名称必须和 Aurora 客户端上 cluster.json 文件中的一项相匹配，这样 Aurora 命令行客户端才知道如何去连接指定的集群。假定配置文件被保存为 outputenv.aurora，运行下面的命令可以将应用部署到集群上：

```
$ aurora job create aurora-cluster/www-data/prod/outputenv outputenv.aurora
```

如果应用已经成功部署，你可以在终端上看到类似输出：

```
 INFO] Creating job outputenv
 INFO] Checking status of aurora-cluster/www-data/prod/outputenv
Job create succeeded: job url=http://mesos-master:8081/scheduler/
www-data/prod/outputenv
```

在 Aurora 客户端的输出中，你可以看到一个作业 URL，点击后可以获取更多关于刚才创建的作业的信息。让我们简要地浏览一下 Aurora 的 Web 接口来获取更多关于这个应用和它在集群中运行的位置。

注意：你可能还记得在第 7 章中，为了保证 OutputEnv 应用可以运行，必须在每一台 Mesos slave 上安装 Ruby 和 Bundler。这是为了说明在集群各个 slave 上要求依赖和将应用依赖打包到一个 Docker 容器镜像时两者的差别。

探索 Web 接口

通过浏览器访问命令行客户端提供的作业 URL 链接，你会看到一个类似图 9.2 的作业配置页面。这个页面提供了这个刚部署的作业的配置概述和作业的每一个运行实例的信息。

如果更新了一个你已经部署的作业的配置，Aurora 会对应用执行滚动升级，确保每个新实例的健康检查通过后才移除老的实例。实例在升级完成前会显示为绿色，所以这个过程通常被称为蓝 / 绿部署，你可以在这个过程中查看这两个版本的配置及之间的差异。

通过点击单个实例 Host 栏的指定链接，你可以打开当前任务所在 Mesos slave 机器上运行的 Thermos 观察者的页面。图 9.3 展示了通过 Thermos 观察者可以看到的一些细节的信息。

图 9.2 作业概述、配置和部署状态

如图 9.3 所示，Thermos 观察者提供了运行中任务和任务配置的相关信息，通过这个页面，你可以查看任务的沙箱、stdout 和 stderr 日志文件，以及所消耗的资源。

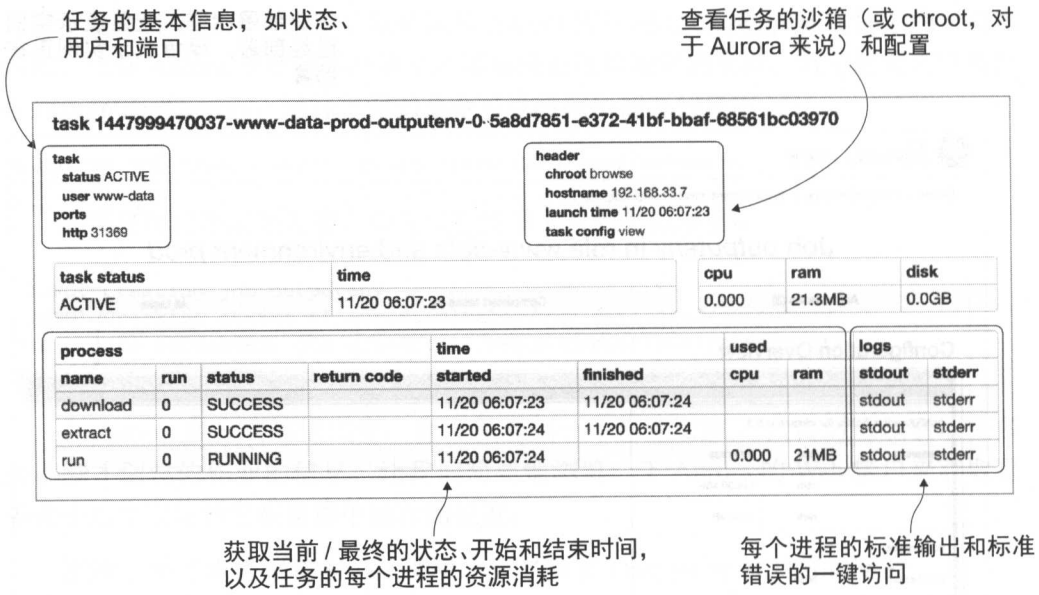


图 9.3 一个特定 Aurora 任务的 Thermos 观察者信息

提示：获取更多服务器配置的信息，请查看用户指南：http://aurora.apache.org/documentation/latest/user-guide。

你会注意到在部署这个应用的时候，集群中的机器需要提前安装一些包：一个或多个 Ruby 的运行环境和 Ruby 用来安装应用依赖包的 Bundler 工具。通过打包你的应用代码和依赖关系到一个独立的容器镜像来避免额外的集群配置和依赖冲突，这种方式在过去几年内让 Docker 愈加热门。

现在你已经在 Aurora 集群上部署了 OutputEnv 应用，让我们看一下怎么在 Aurora 上部署 Docker 容器，以及目前 Aurora 0.9.0 版上关于 Docker 实现的一些局限。

9.3.2 部署基于 Docker 的应用

在撰写本书时，使用 Aurora 部署 Docker 镜像是一个实验性的功能。Twitter 在它们的环境中使用 cgroups，而 Mesos 上对于 Docker 的本地支持也是最近的版本才推出（0.20.0 版）的。一些你在 Marathon 和 Mesos 中获取的基本功能（映射端口，主机和容器之间的挂卷）在这个 Aurora 的版本中还没有实现，但计划在未来版本中发布。

提示：Aurora 可以从 Mesos 上获取整个 `DockerInfo` 字段的信息（镜像、

网络、端口映射，等等），你可以深入了解一下这个功能：https://issues.apache.org/jira/browse/AURORA-1396。

现在，如果你的 Docker 容器需要运行一个侦听网络端口的服务，最好让容器中的服务侦听一个由 Mesos 资源提供的临时端口。你可以在镜像中通过环境变量（如 $PORT）设置侦听端口，或者作为一个参数传入容器中的启动脚本。使用 Aurora DSL 的一种比较简单的实现如下：

```
p = Process(cmdline='./run_server.py --port {{thermos.ports[http]}}')
t = Task(processes=[p])
jobs = [
    Service(
      task=t,
      container=Container(docker=Docker(image='user/image:version'))
    )
]
```

另一种使用 Aurora 运行 Docker 容器的方式是，完全跳过内置的 Docker 功能，而选择直接使用 Docker 命令。毕竟，这也只是另外一个进程。因为这种方式可以让你使用 Docker 中所有构建好的功能，下面的清单演示了如何做到这点。

清单 9.3 使用 Docker CLI 启动一个 Docker 容器

```
docker = Process(
    name='docker-run-nginx',
    cmdline='docker run -p {{thermos.ports[http]}}:80 nginx:1.9'
)
task = Task(
    processes=[docker],
    resources=Resources(cpu=0.1, ram=128*MB, disk=1*GB)
)
jobs = [
    Service(
        cluster='aurora-cluster',
        role='www-data',
        environment='prod',
        name='docker-nginx',
        task=task
    )
]
```

假如清单中的代码保存为 docker-nginx.aurora，你可以运行下面的命令在集群上启动这个容器：

```
$ aurora job create aurora-cluster/www-data/prod/docker-nginx
➥ docker-nginx.aurora
```

> 注意：在这个特殊的例子中，你作为 www-data 用户运行 Nginx 作业，如果这个用户不是 Mesos slave 机器上 docker 组的成员，docker 在执行命令时会报错而无法连接 Docker 守护进程。你需要添加用户到 docker 组，运行下面的命令：sudo usermod -a -G docker www-data。

如果你浏览 Aurora 客户端提供的作业 URL，你会看到 Nginx 容器正在运行，并且 Docker 正把 Mesos 提供的临时端口桥接到 Nginx 容器显示的端口（本例中是 80 端口）。

部署应用（或者服务，对于 Aurora 而言）只是它功能的一半，Aurora 还提供了支持集群上的 Cron 作业调度，就像第 8 章中介绍的 Chronos framework。让我们来看 Aurora 是如何管理计划任务的。

9.4 管理计划任务

不像 Chronos，Aurora 采用了类似于 Cron 作业的传统方法来运行计划作业。它遵循并依赖于熟悉的 Cron 时间表，额外附加一些特性：

- 失败自动重试——如果 Aurora Cron 作业失败了，它会自动重试数次，取决于提供给任务的 `max_task_failures` 属性的值。
- 冲突策略——当上一个作业还没有完成，新的实例又尝试开始时，调度器的行为会被控制。这取决于任务的 `cron_colllision_policy` 属性是被设置为 KILL_EXISTING（杀掉旧作业）还是 CANCEL_NEW（取消新作业）。
- 时区支持——这个集群范围的选项是被指定对 Aurora 调度器而非某个单独的作业。所以默认情况下是 UTC，可以通过使用 `-cron_timezone` 配置选项来设置。

关于 Aurora 里的 Cron 作业的更多详细信息和例子，可以查看官方的在线文档 http://aurora.apache.org/documentation/latest/cron-jobs 。现在，让我们来看一些创建 Cron 作业的基础例子，包括独立运行的和在 Docker 容器内部的。

9.4.1 创建 Cron 作业

在 Aurora DSL 里，除了刚刚介绍的几个 Cron 属性外，创建一个 Cron 作业和创建其他作业是类似的。下面的清单创建了包含 `Process`、`Task` 和 `Job` 的 `Cron` 作业，除了你在 Job 对象中为 `cron_schedule` 属性设置一个值外，其他都与你前面所见的列表相同，让我们看一下。

清单 9.4 创建 Cron 作业

```
sleep = Process(
        name='simple-sleep',
        cmdline="""
        echo "At the tone the time will be: $(date +'%r %Z')"
        echo "Sleeping for 60 seconds."
        sleep 60
        """
)

task = Task(
    processes=[sleep],
    resources=Resources(cpu=0.1, ram=16*MB, disk=1*MB)
)

jobs = [
    Job(
        cluster='aurora-cluster',
        role='www-data',
        environment='prod',
        name='simple-sleep',
        cron_schedule='*/5 * * * *',
        task=task
    )
]
```

假设列表中的代码以 simple-sleep-cron.aurora 的文件保存，你可以通过如下命令创建 Cron 作业：

```
$ aurora cron schedule aurora-cluster/www-data/prod/simple-sleep
➥ simple-sleep-cron.aurora
```

这个命令创建了 Aurora 调度器里的 Cron 作业，并且按照提供的时间表运行作业（每 5 分钟）。Aurora 也允许你通过命令行的客户端手动运行作业。如果你想要按需运行作业，可以使用下面的命令：

```
$ aurora cron start aurora-cluster/www-data/prod/simple-sleep
```

Aurora 客户端中也有额外的 Cron 子命令行，包含移除 Cron 作业和杀掉当前运行中的作业。要了解更多这些额外特性的信息，请执行 `aurora cron -h`。

9.4.2 创建基于 Docker 的 Cron 作业

尽管我在前面提到 Aurora 对 Docker 的支持是实验性质的，但当 Aurora 在 Docker 容器中执行命令时还是相当明确的。借用第 8 章中的一个例子，下面的列表展示了如何在 Docker 容器中执行给定的命令来部署一个 Cron 作业。

列表 9.5 在 Docker 容器中创建 Cron 作业

```
script = 'https://raw.githubusercontent.com/rji/
    ➥ mesos-in-action-code-samples/master/email-weather-forecast.py'

install_python3 = Process(
        name='install_python3',
        cmdline='apt-get update && apt-get -y install python3'
)

download = Process(name='download', cmdline=' '.join(['curl -LO', script]))

run = Process(
        name='run',
        cmdline="""
        export TO_EMAIL_ADDR=user@example.com
        export FROM_EMAIL_ADDR=weather@example.com
        export ZIP_CODE=97201
        export MAIL_SERVER=mail.example.com:25
        export MAIL_USERNAME=weather@example.com
        export MAIL_PASSWORD=ItsTopSecret

        python3 email-weather-forecast.py
        """
)

task = SequentialTask(
    processes=[install_python3, download, run],
    resources=Resources(cpu=0.5, ram=1*GB, disk=768*MB)
)

jobs = [
    Job(
        cluster='aurora-cluster',
        role='www-data',
        environment='prod',
        name='daily-weather-report',
        cron_schedule='0 0 * * *',
        task=task,
        container=Container(docker=Docker(image='python:2.7.10'))
    )
]
```

在这个例子中，我给作业添加了新的 container 属性，指定你定义的 SequentialTask 应该在 Docker Hub 上的 python:2.7.10 的 Docker 镜像中运行。假设列表中的代码以 dailyweather-cron.aurora 的文件保存，你可以通过如下命令在 Aurora 集群中创建 Cron 作业：

```
$ aurora cron schedule aurora-cluster/www-data/prod/daily-weather-report
➥ daily-weather-cron.aurora
```

警告：Aurora 把依赖于 Python 2.7 的 `thermos_executor.pex` 复制入任务沙盒中。如果 Docker 容器中没有 Python 2.7，那么执行器将永远不会注册到 Mesos slave 中，任务则会一直堵塞在 STAGING 的状态中。

希望这些有限的例子能够提供足够的信息可以让你在 Aurora 上开始部署应用和 Cron 作业。和往常一样，Aurora 项目维护了大量的文档，包括用于服务或 Cron 作业上的所有选项和属性的完整列表。更多的信息，可以访问 http://aurora.apache.org/documentation/latest/configuration-reference 。

9.5 管理Aurora

Aurora 是坚持己见的 Mesos framework，它只希望在给定的 Mesos 集群上能运行大部分或者全部的服务。从某种意义上来说，这是件好事：作为系统管理员，它给你单一的方法去维护用户验证和授权，设置资源配额，甚至影响调度的决定，以便 Mesos slave 可以安全地下线进行维护。在接下来的两个小节，针对这些特性我将介绍更多的细节。

9.5.1 管理用户和配额

Aurora 的优势之一是它支持多用户模式，得益于 Twitter 日益增长地用来支持它庞大的工程组织的需求。Aurora 结合了开源的 Apache Shiro 安全 framework 给用户提供身份认证和授权。Shiro 允许使用多个数据源进行身份认证，包含可开箱即用的 LDAP 和 AD 支持（活动目录）。它可插拔的特性也允许你根据自己的实际需求实现专属的身份认证。

在 0.9.0 的版本，Aurora 提供了基础的 HTTP 和 Kerberos 两种认证机制。本节为了简单起见，我把范围限定在基础的 HTTP 认证，以及位于运行 Aurora 调度器机器上的基于 INI 的认证文件。如果你对更复杂的认证和授权方案（例如 Kerberos）感兴趣的话，请访问 http://aurora.apache.org/documentation/0.9.0/security 。

身份认证

基于最简化，Aurora 的安全文件包含一个 Users 部分，在里面你可以这样指定用户名和密码：

```
[users]
alice = topsecretpw
```

注意：Aurora 的安全 ini 文件明文保存了用户的认证信息，请确保设置这

个文件的权限到最小，只允许调度器进程去读取它。在 Aurora 的 bug 追踪频道里有关于在 security.ini 文件中存储 hash 和 salt 化的密码的问题，详情请看 https://issues.apache.org/jira/browse/AURORA-1179 。

要配置基础 HTTP 认证的 Aurora 命令行客户端，用户必须使用 Aurora 的主机名和认证凭证在 ~/.netrc 文件中创建一个条目。

```
machine aurora-cluster.example.com
login alice
password topsecretpw
```

.netrc 文件中的机器名必须和 Aurora 调度器注册到 Mesos master 时使用的名字或者 IP 相匹配。如果在 Aurora 前端有负载均衡器，或者通过 DNS 域名指向到该调度器，那么你可以在启动调度器的时候，使用 `-hostname` 的选项来设置。

提示：关于 .netrc 文件的更多信息，请查看 http:// www.gnu.org/software/inetutils/manual/html_node/The-_002enetrc-file.html 。

授权

除了用户身份认证，Aurora 也允许你配置用户可能属于的任意角色。这些角色都基于 Aurora Thrift API 提供的可用的角色。具体的 API 文档可以通过访问 Aurora 调度器来获取：http://aurora.example.com:8081/apiclient/api.html。

在下面的例子中，用户 Alice 被授予了对 Aurora 集群的管理员级别的访问权限，用户 Bob 则只被授予了 `accounting` 角色的访问权限，而用户 Carol 则没有被加到任何角色中：

```
[users]
alice = secret, admin
bob   = secret, accounting
carol = secret

[roles]
admin      = *
accounting = thirft.AuroraAdmin:setQuota
```

管理资源配额

在 Aurora 中，生产级别的任务可以取代非生产的任务。当需要额外的集群资源时，一个生产任务可以杀死低级别的非生产任务。在 Aurora 内部可以有多套环境（开发、生产演练、生产以及其他），它们能安全地运行在同个集群内，并且不会有对生产负载造成冲击的危险。

资源配额按生产作业的要求，并储备集群资源池为以后运行作业所用。Aurora的管理客户端中有两条子命令行用来设置和增加配额：

```
aurora-admin set_quota <role> <cpus> <mem> <disk>
aurora-admin increase_quota <role> <cpus> <mem> <disk>
```

如果集群的安全是像《身份认证和授权》这一节所说的一样设置的话，那么你需要成为admin或者accounting的角色才能修改资源配额。

作为Aurora的用户，你能通过使用Aurora的用户客户端，来决定某个指定角色分配的资源，以及确认生产和非生产资源的消耗情况：

```
aurora quota get devcluster/example_role
```

9.5.2 执行维护

Aurora中有个管理特性是可以在集群中的一组机器上执行调度器级别维护的。它让你可以把某台服务器设置为维护模式，停掉上面的任务并且在集群其他的节点上重新调度。执行维护，以及让这台机器继续处理正常的服务。

Aurora的管理客户端显示了这个功能性，其包含以下的子命令。

- host_deactivate——让一台或者一组服务器进入维护模式。它会高效地让Aurora在做调度决策时将服务器降级，但取决于集群的容量，任务仍有可能会被调度到该失活服务器上。
- host_drain——杀掉一台或者一组服务器上运行中的任务，并且阻止新的任务再被调度到该服务器上。这个子命令最好和host_deactivate组合使用：你可以对大批量的服务器进行调度降级，然后对使用host_drain其中小批量的服务器执行批量的维护。
- host_activate——解除一台或者一组服务器的维护模式，并恢复正常的调度。
- host_status——获取一台或者一组服务器的维护状态，包括SCHEDULED (host_deactivate)，DRAINED (host_drain)或者NONE (host_activate)。

使用Aurora的管理客户端，每个子命令都可以使用下面的格式配合指定的集群运行：

```
$ aurora-admin <subcommand> --hosts=host.example.com[, ...] <cluster_name>
```

某些子命令有额外的选项，你会发现它们很有用。例如，这里提到的每个子命令可以把主机的文本文件带入维护模式。并且host_drain子命令可以接受额外的参数，当主机上面所有的任务都停止后才来执行某个脚本。关于管理客户端的完

整用法，请执行 `aurora-admin help <subcommand>` 。

9.6 小结

在本章中你学习了通过 Mesos 上的 Apache Aurora framework 来部署应用以及计划作业。下面有是需要记住的几点：

- Aurora 由四个主要的组件组成：调度器，执行器，观察者以及客户端。执行器和观察者都是 Aurora 发行版本里 Thermos 项目内部的组件。
- Aurora 调度器负责接受或者拒绝来自 Mesos master 的资源提供。它是高度可配置的，并且通过 Mesos slave 上面 Thermos 执行器来启动任务。
- Thermos 观察者是运行在 Mesos slave 上的服务，它拉取运行中的执行器的任务信息，并提供一站式访问任务的日志、沙盒以及资源消耗统计的服务。
- Aurora 的用户客户端通常是发布给个人用户或者工程师来管理 Aurora 集群上服务和 Cron 作业。关于完整的使用方法，请执行 `aurora -h`。
- Aurora 的管理客户端通常是发布给需要维护集群的管理员来设置和修改用户配额和执行主机维护。关于完整的使用方法，请执行 `aurora-admin -h`。
- 当在 Docker 容器内部运行服务或者 Cron 作业时，请确保 Python 2.7 是安装好的，这样 Thermos 执行器才能注册到 Mesos slave 上。否则，任务将被阻塞在 `STAGING` 状态。
- Aurora 使用 Apache Shiro 安全 framework 来实现用户认证和授权。Aurora 提供非认证的请求、简单 HTTP 认证和 Kerberos 认证的支持。
- 可基于每个角色设置生产级别的资源配额，在必要情况下，它会抢占非生产的任务。
- 使用 Aurora 的管理客户端，你可以让主机进入维护模式来影响集群中部分主机的调度决策。使用提供的各种子命令，你可以在执行维护前停掉主机上的任务，在基础架构维护结束后让主机恢复正常服务。

因为 Aurora 是强大的，并且扩展到成千上百用户的 framework，在本章中我不能涵盖所有的特性和配置。幸运的是，你可以查阅 Aurora 项目提供的详尽文档 http://aurora.apache.org/documentation/0.9.0 。

第 10 章（也是最后一章）将提供 Mesos API 的初始版本，以及如何开始开发你专属的 Mesos framework。

10 framework开发

本章内容

- 构成 mesosframework 的组件
- mesos 调度器和执行器 APIs
- 使用 Python 为 Mesos 编写 framework

欢迎来到 *Mesos* 实战的最后一章。到现在为止，我已经介绍了诸如 Mesos 如何为数据中心计算提供一种全新的架构，如何部署 Mesos，以及如何部署应用和 Cron 任务等主题。本章将介绍如何入门开发第一个 Mesos framework。

不同于前面几章，比起之前介绍的操作方面的内容，本章内容更侧重于开发方面。在我们开始之前，假定你们已经具备一定的软件开发经验以及阅读、编写 Python 代码的能力。尽管 Mesos 也提供了对其他语言的支持，如 C ++、Java 和 Scala，也有社区贡献的脚本语言如 Erlang 和 Go 等语言，但由于 Python 较其他语言更容易阅读和理解，所以本书将使用 Python。

本章介绍了构成 Mesos framework 的组件，并指出编写 framework 时需要考虑的注意事项。由于 Mesos API 提供了大量的功能，所以我们在本章中不可能涵盖所有的内容。学习完本章，你可以更好地了解 Mesos API，以及能够构建和运行这里

的示例代码。

准备好了吗？让我们开始下面的学习。

10.1 framework基础

Mesos framework 是使用 Mesos 接收集群资源的分布式系统。Mesos 中 *framework* 这个词是指任何在 Mesos master 中注册并运行在 Mesos 集群之上的应用程序。Mesos framework 包含两个组件——调度器和执行器：

- 调度器——注册到 Mesos master 中，执行接受或拒绝来自 master 的资源提供的逻辑流程，并决定哪些任务使用哪些资源。
- 执行器——注册到 Msos slave 中，执行管理任务生命周期的逻辑流程，包括任务启停和提供状态更新。当一个进程首次启动时，执行器会发送一次状态更新消息表明该任务正在运行；当它退出时，执行器会发送一次更新表明该任务已完成或失败。

就最基本的形式而言，图 10.1 描述了 Mesos、调度器和执行器之间的交互关系。

注意：本章提供了开发自定义调度器和执行器的相关信息。值得注意的是，在开发 Mesos framework 时，你不一定需要自己编写自定义执行器，完全可以使用 Mesos 内置的 `CommandExecutor` 在 shell 下执行命令。我会在本章稍后讨论这两种方法。

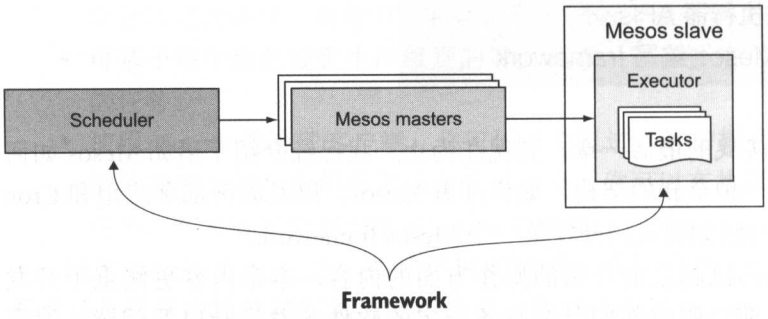


图 10.1 组成 Mesos framework 的调度器和执行器，运行在 Mesos master 和 slave 的环境中

让我们思考一下在调度器环境中获取资源提供的生命周期。在图 10.2 中，可以看到调度器通过启动执行器去获取并使用 Mesos slave 提供的可用资源的周期性过程。

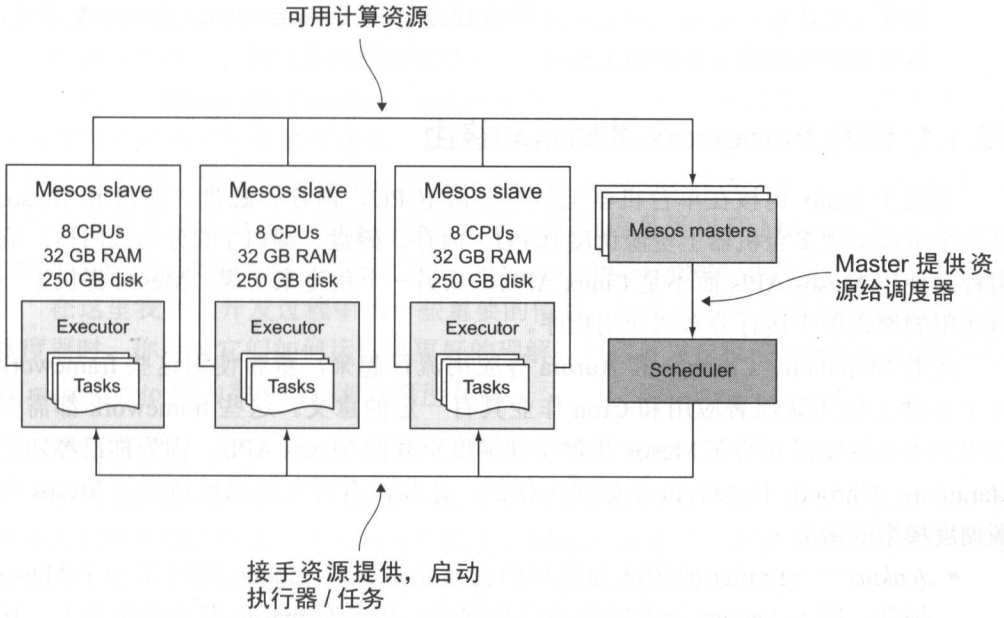


图 10.2 调度器基于资源提供启动执行器和任务

当 Mesos slaves 将可用的资源信息通告给 master 的时候，master 会根据公平共享资源的原则做出决策，将资源分配给已注册的 framework。这些资源接着会被提供给 framework 中负责接收或拒绝资源逻辑的调度器。如果资源提供被接受，调度器负责启动 slave 中的执行器，进而启动 framework 任务。

除了调度器和执行器，还有第三个需求，尽管该需求（潜在地）独立于 framework 本身：你需要一个方法来提供自定义执行器代码以供 Mesos 使用。幸运的是，有如下几个选项供你选择：

- *HTTP*——如果调度器可以运行 Web 服务，你可以将执行器的代码绑定到调度器中，并在 HTTP 上提供服务。Mesos 提取器在尝试调用执行器之前就可以通过 http 请求将代码下载到沙盒中。
- 网络文件系统——Mesos 提取器同样支持从 FTP 服务器或者 Amazon S3 服务中下载执行器代码。
- 分布式文件系统——使用分布式文件系统，比如 HDFS，你可以使执行器独立于调度器实现分布式化，降低它们之间的耦合关系。
- 使用 slave 的本地文件系统或者一个挂载点——如果你已经有了配置管理工具，如 Puppet 或 Chef 或者在集群中每个 slave 的普通挂载点（比如 NFS export 挂载点），你可以引用执行器的代码作为磁盘上的路径。如果你决定使用这种途径，你需要根据执行器的绝对路径来引用（在所有主机必须保持

相同）或者为 `mesos slave` 守护进程设定 `--frameworks_home` 配置项，然后使用指向执行器的相关路径。

10.1.1 编写 framework 的时机和缘由

类似于 Linux 内核在单台机器上调度资源（CPU、内存、磁盘、端口），Mesos 是一个可以实现多台机器上资源调度（CPU、内存、磁盘、端口）的分布式内核。应用程序使用 Mesos APIs 而不是 Linux APIs。从另一个角度来考虑，Mesos 提供了一组通用的原语用于运行分布式应用程序。

随着 Marathon、Chronos 和 Aurora 等应用流行起来，重新使用这些 framework 对于各种工程团队部署应用和 Cron 作业具有一定的意义。这些 framework 都需要使用到本章将要提到的在 Mesos 集群实现调度负载的 Mesos APIs。因为你已经知道 Marathon、Chronos 和 Aurora 是如何工作的，让我们看看一些已经适应了 Mesos 资源调度模型的服务：

- *Jenkins*——Twitter 的团队为流行的持续集成系统 Jenkins 编写了一个 Mesos 插件。因为 Jenkins 分发它的项目构建在一组具有 master 节点的机器上，这与 Mesos 资源调度模型十分匹配。Mesos 插件的好处是什么呢？Jenkins 现在可以调度 CPU、内存、磁盘等它所需要与其他 framework 在更大 Mesos 集群上一同运行的资源，而不需要它自己的静态分区集群。
- *HDFS*——Mesosphere 的团队维护许多 Mesos framework，包括在 Mesos 之上运行 Hadoop 分布式文件系统（HDFS）。通常来说，Hadoop 集群（类似 Jenkins）需要运行在自己的机器集群上。确保 HDFS 的 namenode 在 `DataNodes` 之前启动，而且每个实例必须在不同 Mesos slave 上运行的逻辑都可以构建进调度器中，并且 HDFS 可以使用庞大的 Mesos slave 集群上的所有可用的存储。Mesos 提供了开箱即用的原语，诸如容错性和高可用性等，确保当有节点出现异常的时候，HDFS 节点可以在正常的机器上重新调度。

如果你打算部署应用程序和调度作业到一个 Mesos 集群上，你可能不需要开发自己的 framework，而是选择使用一个现成的解决方案。毕竟，开发一种新的分布式系统不是一个简单的任务，而创造一个产品级 framework 甚至更困难。但是如果你想尝试编写一个新的 Mesos-native 应用程序，移植已有的服务到 Mesos 上，或者你的应用程序需要特殊的调度逻辑，那你最好的做法是编写你自己的 framework。

10.1.2 调度器的实现

作为 framework 组件，调度器主要负责如下功能：

- 接收来自 master 的资源提供；

- 接受资源提供并启动任务，或拒绝提供；
- 接收和响应来自任务的状态更新。

本质上，Mesos 通过 *protocol buffers* 系统——一个最初在谷歌开发的项目，序列化组件之间的消息来实现通信。尽管对 protocol buffers 的详细介绍已经超出了本书的内容范畴，但使用 protocol buffers 系统允许 Mesos 开发人员可以用多种语言编写实现符合 Mesos API 的通用消息格式。这些信息可以在 Mesos 代码库 https://github.com/apache/mesos/blob/0.22.2/include/mesos/mesos.proto 中找到。

在这里我介绍开发过程中的一些重要的信息，让你在准备好编写并运行自己的调度器时，你会对它们如何运作有更好的理解。当你开发自己的 framework 的时候，以 mesos.proto 作为参考应该对你有所帮助。

资源提供

因为 Mesos 负责在多个 framework 中调度集群的计算资源，所以首先我们要了解资源提供本身的格式。提供的消息描述了 Mesos slave 中可用的资源总量。除了列出可用的资源，还包括资源所在的主机名和 Mesos slave 的唯一 ID：

```
message Offer {
  required OfferID id
  required FrameworkID framework_id
  required SlaveID slave_id
  required string hostname
  repeated Resource resources
  repeated Attribute attributes
  repeated ExecutorID executor_ids
}
```

还记得第 4 章中我提到的，通过三种不同的类型：scalars、ranges 和 sets，在指定的 Mesos slave 中定制资源。资源的信息作为提供的一部分，用以描述这些资源类型：

```
message Resource {
  required string name
  required Value.Type type
  optional Value.Scalar scalar
  optional Value.Ranges ranges
  optional Value.Set set
  optional string role [default = "*"]
}
```

由于在 Mesos 中所有调度取决于资源提供，所以 `resourceOffers()` 方法是为数不多的你需要在自己的调度器中实现的方法。如下就是你开始实现调度逻辑的地方：

```
def resourceOffers(self, driver, offers):
    for offer in offers:
        logging.info("Received offer with ID: {}".format(offer.id.value))
        logging.info("Declining offer ID {}".format(offer.id.value))
        driver.declineOffer(offer.id)
```

前面的代码示例打印了日志信息和拒绝他们收到的所有资源提供。通常，你会检查队列里待处理的工作来确定资源是否可用和是否满足将要启动任务的需求。10.2节将更加详细地介绍resourceOffers()和组成调度器API的其他详细方法。

Framework信息

FrameworkInfo message负责描述framework的信息。在你创建一个framework时，只有两个选项必须指定：即将运行framework的用户和framework的名称：

```
message FrameworkInfo {
  required string user
  required string name
  optional FrameworkID id
  optional double failover_timeout [default = 0.0]
  optional bool checkpoint [default = false]
  optional string role [default = "*"]
  optional string hostname
  optional string principal
  optional string webui_url
}
```

FrameworkInfo的其他选项包括调度器所注册的主机名，用于识别framework的主体，还有调度器网页界面的URL(如果有的话)。如果你为framework的webui_url参数指定一个值，那么你可以从Mesos UI中快速导航到调度器的网页界面。

在Python中，你将创建一个新的FrameworkInfo()实例来设置这些参数选项，然后将实例传递给MesosSchedulerDriver，MesosSchedulerDriver的更多细节将在下一节中介绍：

```
framework = mesos_pb2.FrameworkInfo()
framework.user = ''
framework.name = 'ExampleFramework'
framework.checkpoint = True
framework.principal = 'ExampleFramework'
...
driver = MesosSchedulerDriver(..., framework, ...)
...
```

如果你指定 framework.user 参数为空字符串（就像我之前做的），Mesos 会假定你打算使用当前的系统用户来运行 framework。

调度器驱动

MesosSchedulerDriver 负责连接 Mesos master 并与之进行交互。在实例化驱动程序之后，你将使用它来接受资源提供，并通过调用 launchTasks() 方法在 slave 中启动任务，或者当没有工作要处理时，通过调用 declineOffer() 方法拒绝提供。

```
class ExampleScheduler(mesos.interface.Scheduler):
    ...
    def resourceOffers(self, driver, offers):
        for offer in offers:
            driver.declineOffer(offer.id)
    ...
driver = MesosSchedulerDriver(ExampleScheduler(), framework, master)
driver.run()
```

10.2 节将详细介绍组成 Scheduler API 的各种方法和 SchedulerDriver。

10.1.3 执行器的实现

组成 Mesos framework 的另一个组件就是执行器。简而言之，执行器负责如下功能：

- 启动和停止任务；
- 给调度器提供任务状态更新信息。

为了做到开箱即用，Mesos 中包含了一个内嵌的 CommandExecutor 执行器，用于给 /bin/sh 运行命令参数。就像在你的笔记本中执行命令或者通过服务器的控制台直接执行命令一样，在本节中，我将介绍如何实现 Mesos 自定义执行器。

由于 shell 可以启动大多数的进程，你可能想知道为什么需要为你的 framework 编写一个定制执行器。通常情况下，当你需要对你所启动的任务执行健康检查时，当你想要运行一些用你的 framework 程序语言编写的代码时，或者当你的进程需要更多的设置而不仅仅是在 shell 中运行命令那么简单时，这样做是一个好主意。基于健康检查的场景下，你可以根据健康检查的状态将任务的状态更新信息发送反馈给调度器，从而允许调度器在处理特殊情况时能够作出进一步的决策。

因为健康检查是第一个想到的使用案例，而且能很好地映射各种任务状态，所以让我们从 TaskStatus 消息开始讲起吧。

执行器任务状态

Mesos 执行器使用 TaskStatus 消息更新任务的状态。任务的状态更新需要至少指定任务 ID（task_id）和状态：

```
message TaskStatus {
  ...
  required TaskID task_id
  required TaskState state
  optional string message
  optional Source source
  optional Reason reason
  optional bytes data
  optional SlaveID slave_id
  optional ExecutorID executor_id
  optional double timestamp
  optional bytes uuid
  optional bool healthy
}
```

由于 Mesos 的多个组件都需要用到任务的状态，所以在 mesos.proto 中单独定义了 TaskState message 来存放一个任务可能具有的状态：

```
enum TaskState {
  TASK_STAGING
  TASK_STARTING
  TASK_RUNNING
  TASK_FINISHED
  TASK_FAILED
  TASK_KILLED
  TASK_LOST
  TASK_ERROR
}
```

第一个状态 TASK_STAGING，只能由 Mesos 内部使用而不能够由 framework 状态更新使用。此外，在 TASK_RUNNING 后的每个状态（例如，TASK_FINISHED，TASK_FAILED 等）都被认为是一个终止状态。这意味着任务不再运行，以及 Mesos 应该清理任务结束后的数据，包括调度任务的沙盒来实施垃圾收集和提供之前消耗的资源给另一个任务或 framework 使用。

任务 ID 在执行器中的 launchtask() 方法中使用到，详细内容将在第 10.3 节中讨论。到现在为止，你可以从的代码片段中了解到执行器启动任务和提供任务状态更新的基本实现：

```
def launchTask(self, driver, task):
    update = mesos_pb2.TaskStatus()
    update.task_id.value = task.task_id.value
    update.state = mesos_pb2.TASK_RUNNING
    driver.sendStatusUpdate(update)
    ...
```

在发送初始化状态 TASK_RUNNING 更新给驱动程序后，接着执行运行任务所需的所有逻辑，并发送其他额外需要的状态信息。

定义调度器使用的执行器

虽然 framework 中的调度器和执行器之间是松耦合的，但调度器还是需要获取一些关于启动其任务的执行器信息。ExecutorInfo 包含如执行器 ID、待执行的命令以及将要分配的资源等信息：

```
message ExecutorInfo {
  required ExecutorID executor_id
  optional FrameworkID framework_id
  required CommandInfo command
  optional ContainerInfo container
  repeated Resource resources
  optional string name
  optional string source
  optional bytes data
  optional DiscoveryInfo discovery
}
```

作为 ExecutorInfo 消息的一部分，CommandInfo 主要负责描述如何执行一个任务：

```
message CommandInfo {
  message URI {
    required string value = 1;
    optional bool executable = 2;
    optional bool extract = 3 [default = true];
  }
  message ContainerInfo {
    required string image = 1;
    repeated string options = 2;
  }
  optional ContainerInfo container = 4;
  repeated URI uris = 1;
  optional Environment environment = 2;
  optional bool shell = 6 [default = true];
  optional string value = 3;
  repeated string arguments = 7;
  optional string user = 5;
}
```

当使用自定义的执行器启动任务时，首先需要根据ExecutorInfo信息实例化一个新的执行器，并将它传递给你的调度器实例：

```
executor = mesos_pb2.ExecutorInfo()
executor.executor_id.value = 'ExampleExecutor'
executor.command.value = os.path.abspath('./example-executor.py')
executor.name = 'Example Executor'
...
driver = MesosSchedulerDriver(ExampleScheduler(executor) ...)
```

通过将执行器传递给调度器实例之后，调度器在创建任务的时候就可以获得访问权限（通过TaskInfo）。

注意：在这个例子中，以及本章的其他地方，已假设我们的开发环境是同时运行了Mesos master和salve的机器，我们在这样的开发环境中做framework的开发。因此，调度器和执行器都可以在同一个文件系统上使用。你可能注意到了Commandinfo消息中的uris字段；使用Mesos提取器时，请查看executor.command.uris.add()方法。这个方法将允许你使用本章开头描述的各种方法下载执行器的代码到沙盒中。

任务信息

TaskInfo消息用于描述一个任务，它和我之前提到的其他消息有点不同。任务信息从调度器传递给执行器（通过driver.launchTasks ()）。当它用自定义执行器运行时，需要设置ExecutorInfo，或者当它使用内置的Mesos CommandExecutor时，需要设置CommandInfo：

```
message TaskInfo {
  required string name
  required TaskID task_id
  required SlaveID slave_id
  repeated Resource resources
  optional ExecutorInfo executor
  optional CommandInfo command
  optional ContainerInfo container
  optional bytes data
  optional HealthCheck health_check
  optional Labels labels
  optional DiscoveryInfo discovery
}
```

当调度器接收了一个资源提供后会构建一个任务并将其发送给MesosSchedulerDriver，这样一个任务就创建完毕了。

```
def resourceOffers(self, driver, offers):
    for offer in offers:
        task = mesos_pb2.TaskInfo()
        task_id = str(uuid.uuid4())
        task.task_id.value = task_id
        task.slave_id.value = offer.slave_id.value
        task.name = "task {}".format(task_id)
        task.executor.MergeFrom(self.executor)

        cpus = task.resources.add()
        ...

        mem = task.resources.add()
        ...

        driver.launchTasks(offer.id, [task])
```

当一个任务在 Mesos 集群中启动的时候，它通过启动自定义执行器来执行你的代码，让你可以对服务的健康检查和状态更新有更多的控制和管理。

更多的调度器和执行器的消息类型的定义可以到 mesos.proto 文件中查看，但到目前为止，你应该对 Mesos 如何为 framework 提供资源和启动任务的过程有所了解了。在开发的过程中，你可以参考 mesos.proto 和 Python 的 Mesos API 手册，或者使用 IntelliJ IDEA 或 PyCharm 等来自 JetBrain 的 IDE 工具来帮助你完成开发。

在下一节中，在学习开发你自己 framework 的过程中，你将应用你目前所学习到的知识。

10.2 调度器开发

正如本书前面所学到的，Mesos 遵循如下两层调度模型：Master 提供资源给调度器，然后调度器根据是否有工作（要启动的任务）要执行来决定接受或拒绝提供。调度器可以利用提供给它的信息决定何时何地调度指定工作负载；在 HDFSframework 的例子中，调度器可以决定 HDFS `DataNode` 必须运行在特定的 Mesos slave 上。这样确保 HDFS 所有 `DataNode` 的任务不能在一个节点上运行，避免单点故障风险。

最后一节中会提到调度器和执行器是通过一个抽象的驱动连接到 Mesos 中的，有一个独立的驱动程序供调度器（被称为 `Schedulerdriver`）和执行器（被称为 `Executordriver`）使用。在图 10.3 中，你可以看到在 slave 中调度器是如何以一个新的进程来启动任务的。

下面的章节内容将会用代码演示如何使用 Mesos 的 API 和 Mesos 中可用的驱动程序去启动自定义调度器和执行器，并在集群上运行任务。为简便起见，正文只记录实现调度器工作最少的可行的方法实现。本书的补充代码包含了更完整的例子。

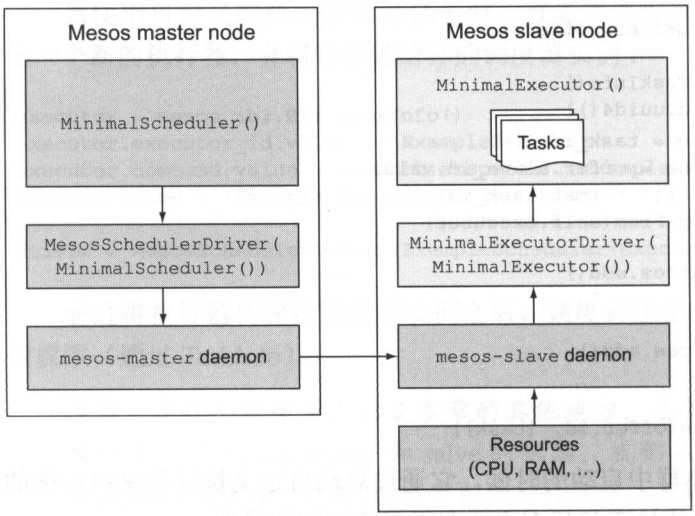


图 10.3 调度器、调度器驱动程序、执行器和执行器驱动程序之间的交互关系

10.2.1 使用调度器 API

当你开发自定义调度器时，你可以使用 Mesos 接口预定义的若干方法。这些方法我称之为调度器 *API*，允许你可以接受 Mesos 提供的默认逻辑，或者通过子类化 `mesos.interface.scheduler` 和重构其中的方法来实现你自定义的逻辑。

你可以通过检查 Mesos 源代码[1]来浏览如何使用所有接口及每个方法的详细描述。为了获取完整的工作 framework，作为一个起点，在开发自己的 framework 之前，你可以查看第 10 章目录下的例子。

调度器 API 概况

调度器 API 中的每个方法都粗略地解释其作用以及数据作为参数传入方法。借用 Mesos 的 Python 接口源代码，Scheduler API 中可用的方法及其参数如下所示：

```
class Scheduler(object)
  def registered(self, driver, frameworkId, masterInfo)
  def reregistered(self, driver, masterInfo)
  def disconnected(self, driver)

  def resourceOffers(self, driver, offers)
  def offerRescinded(self, driver, offerId)
  def statusUpdate(self, driver, status)
  def frameworkMessage(self, driver, executorId, slaveId, message)
```

1 https://github.com/apache/mesos/blob/0.22.2/src/python/interface/src/mesos/interface/__init__.py#L34-L129

```
def slaveLost(self, driver, slaveId)
def executorLost(self, driver, executorId, slaveId, status)
def error(self, driver, message)
```

当一个指定的事件发生时，不管调度器是否有registered()，reregistered()，或接收frameworkMessage()等方法，API允许你接受默认的逻辑或实现自己的逻辑。你唯一需要实现的方法就是resourceOffers()。

编写自己的调度器

以下列出你开发Mesos自定义调度器的最简单的例子。同时该清单还列出如何使用你下个章节将会实现的自定义执行器的信息。

表10.1 开发一个最小的Mesos调度器

```
from __future__ import print_function
import sys
import uuid
from threading import Thread
from mesos.interface import Scheduler, mesos_pb2
from mesos.native import MesosSchedulerDriver

class MinimalScheduler(Scheduler):
    def __init__(self, executor):
        self.executor = executor

    def resourceOffers(self, driver, offers):
        for offer in offers:
            task = mesos_pb2.TaskInfo()
            task_id = str(uuid.uuid4())
            task.task_id.value = task_id
            task.slave_id.value = offer.slave_id.value
            task.name = "task {}".format(task_id)
            task.executor.MergeFrom(self.executor)
            task.data = "Hello from task {}!".format(task_id)

            cpus = task.resources.add()
            cpus.name = 'cpus'
            cpus.type = mesos_pb2.Value.SCALAR
            cpus.scalar.value = 0.1

            mem = task.resources.add()
            mem.name = 'mem'
            mem.type = mesos_pb2.Value.SCALAR
            mem.scalar.value = 32

            tasks = [task]
            driver.launchTasks(offer.id, tasks)
```

通过把mesos.interface.Schedueler划入子集来实现一个新的集合

接受一个ExecutorInfo()对象作为参数

在MinimalScheduler集合里你必须执行resourceOffer()方法

一个或多个可用的资源提供作该为方法的数组

用CPUs、内存、执行器、名字和数据构建一个新的TaskInfo()对象

使用特定的资源提供来启动一个或多个任务

```
def main():
    executor = mesos_pb2.ExecutorInfo()
    executor.executor_id.value = 'MinimalExecutor'
    executor.name = executor.executor_id.value
    executor.command.value = '/path/to/executor-minimal.py'

    framework = mesos_pb2.FrameworkInfo()
    framework.user = ''
    framework.name = 'MinimalFramework'
    framework.checkpoint = True
    framework.principal = framework.name
```

创建一个新的ExcutorInfo()任务

创建一个新的FramworkInfo()任务

在前面的例子中，调度器接受它所收到的每个资源提供，并使用资源提供启动一个或多个任务，每个任务都分配 0.1 个 CPUs 和 32MB 内存。在定义完调度器逻辑后，需要把 `main()` 中的执行器和 framework 对象作为参数传递给 `MesosSchedulerDriver()`，由它负责与 Mesos master 进行通信。下一节我将介绍 `SchedulerDriver`。

10.2.2 使用 SchedulerDriver

Mesos 调度器驱动程序为 Mesos master 获取调度器代码提供了接口。它是用来做以下的事情：

- 通过使用 `run()`,`stop()` 等，管理调度器生命周期。
- 通过使用 `launchTasks()`，`declineOffer()` 等与 Mesos 进行交互。

为简短起见，本章没有涵盖调度器驱动程序的所有细节。你可以通过阅读 Mesos Python bindings[1] 中的相关代码了解各种接口中各种方法的详细使用方法。

```
class SchedulerDriver(object)
  def start(self)
  def stop(self, failover=False)
  def abort(self)
  def join(self)
  def run(self)
  def requestResources(self, requests)
  def launchTasks(self, offerIds, tasks, filters=None)
  def killTask(self, taskId)
  def declineOffer(self, offerId, filters=None)
  def reviveOffers(self)
  def acknowledgeStatusUpdate(self, status)
  def sendFrameworkMessage(self, executorId, slaveId, data)
  def reconcileTasks(self, tasks)
```

清单 10.2 提供了一个调用调度器驱动程序的最简单的例子，它负责将调度器注

1 https://github.com/apache/mesos/blob/0.22.2/src/python/interface/src/mesos/interface/__init__.py#L132-L244

册到 master 中。如果运行调度器的机器可以访问 Mesos master，那么调度器驱动程序可以在任何地方运行，反之亦然。调度器驱动程序需要几个参数，包括调度器子类的实例，framework 信息和要连接的 Mesos master。

清单 10.2 用调度器驱动程序连接调度器到 Mesos

```
def main():
    ...
    driver = MesosSchedulerDriver(
        MinimalScheduler(executor), framework, 'localhost:5050')

    status = 0 if driver.run() == mesos_pb2.DRIVER_STOPPED else 1
    driver.stop()
    sys.exit(status)
```

创建一个新的 MesosSchedulerDriver.

运行 MesosSchedulerDriver,

停止驱动程序。

值得注意的是，调度器驱动程序在进一步的执行过程中会阻塞，因此最好是用它自己的线程来运行驱动程序。本书的 GitHub 库提供的例子说明了如何实现这种方式。

10.3 执行器开发

当你在开发自定义 Mesos framework 时，并不一定需要编写自定义执行器。相反，在创建任务的时候，你的调度器可以通过修改 `TaskInfo()` 来使用 Mesos 内嵌的 `CommandExecutor`，如下面这样：

```
task.command.value = 'echo "Hello, world!" && sleep 30'
```

`CommandExecutor` 接收命令并追加到 /bin/sh –c 执行，这种方法在很多情况下是可行的。但是为了内容的完整性，在本章中我们还是会介绍执行器 API 的。

10.3.1 使用执行器 API

Mesos 执行器接口预定义了开发自定义执行器时需要用到的若干方法。*执行器 API* 中的方法，允许你可以接受 Mesos 提供的默认逻辑，也可以通过子类化 mesos.interface.Executor 实现自己的逻辑。

你可以通过 Mesos 源代码[1] 浏览如何使用所有接口及每个方法的详细描述。为了获取完整的工作 framework，作为一个起点，在开发自己的 framework 之前，你可以查看第 10 章目录下的例子。

1 https://github.com/apache/mesos/blob/0.22.2/src/python/interface/src/mesos/interface/__init__.py#L246-L310

Executor API 概述

Executor API 粗略地解释每个方法的作用以及方法参数的意义。借用 Mesos 的 Python 接口源代码，如下所示为 ExecutorAPI 中可用的方法及其参数：

```
class Executor(object)
  def registered(self, driver, executorInfo, frameworkInfo, slaveInfo)
  def reregistered(self, driver, slaveInfo)
  def disconnected(self, driver)
  def launchTask(self, driver, task)
  def killTask(self, driver, taskId)
  def frameworkMessage(self, driver, message)
  def shutdown(self, driver)
  def error(self, driver, message)
```

当一个指定的事件发生的时候，不管执行器是否调用 `registered()` 或通过 `disconnected()` 断开 Mesos slave 链接，API 都允许你接受 Mesos 默认的处理逻辑或者实现自己的逻辑。

编写自定义执行器

下列清单中提供了一个实现自定义 Mesos 执行器的最简单的案例。在执行器注册后，不同任务可重复使用该执行器；注意：我们执行器部分的程序打印出的 `task.data`，它是从调度器中通过所有方式传递过来的任意数据。

清单 10.3 开发最小化 Mesos 执行器

```
from __future__ import print_function
import sys
import time
from threading import Thread
from mesos.interface import Executor, mesos_pb2
from mesos.native import MesosExecutorDriver

class MinimalExecutor(Executor):
    def launchTask(self, driver, task):
        def run_task():
            update = mesos_pb2.TaskStatus()
            update.task_id.value = task.task_id.value
            update.state = mesos_pb2.TASK_RUNNING
            driver.sendStatusUpdate(update)

            print(task.data)
            time.sleep(30)

            update = mesos_pb2.TaskStatus()
            update.task_id.value = task.task_id.value
```

通过子类化 mesos.interface.Executor 来实现一个新的集合

执行 lauchTasks() 方法来运行任务

为任务返回到调度器提供一个状态更新信息

用户定义代码通常在此处出现

```
        update.state = mesos_pb2.TASK_FINISHED
        driver.sendStatusUpdate(update)

    thread = Thread(target=run_task, args=())
    thread.start()
```

任务应在它自己的线路或进程中运行。

在前面的示例中，我们在一台主机上创建执行器并运行一个给定的任务。当任务最初创建的时候，执行器会打印出从调度器中通过所有方式传递过来的 task.data 值。但为了连接执行器和 mesos slave，你需要调用执行器驱动程序。

10.3.2 使用执行器驱动程序

Mesos 执行器驱动程序为执行器代码连接到 Mesos slave 提供了接口。它是用来做以下事情：

- 通过 `run()`，`stop()` 等管理执行器生命周期。
- 通过 `launchtask()` 启动任务，并使用 `taskstatus()` 提供状态更新的信息。

你可以在 Mesos 源码库[1]了解接口中各种方法的详细作用和原理。现在让我们先了解一下驱动程序提供的各种方法：

```
class ExecutorDriver(object)
  def start(self)
  def stop(self)
  def abort(self)
  def join(self)
  def run(self)
  def sendStatusUpdate(self, status)
  def sendFrameworkMessage(self, data)
```

在下面的代码片段中，我提供了一些可以调用执行器驱动程序的样例代码，由它负责将执行器注册到 Mesos slave 中。作为执行器代码的一部分，这意味着当调度器启动一个任务的时候，它会在 Mesos slave 中执行。调度器驱动程序需要一个自定义执行器的实例作为参数才能运行：

```
if __name__ == '__main__':
    driver = MesosExecutorDriver(MinimalExecutor())
    sys.exit(0 if driver.run() == mesos_pb2.DRIVER_STOPPED else 1)
```

创建一个新的 MesosExecutorDriver 实例

运行驱动程序

1 https://github.com/apache/mesos/blob/0.22.2/src/python/interface/src/mesos/interface/__init__.py#L314-L367

在以最低的要求实现自定义调度器和执行器后，让我们看看如何在开发环境中测试我们的 framework 代码。下一节将介绍常见的基于 Vagrant 的开发环境以及创建可用于生产环境的 framework 时需要注意的一些事项。

10.4 运行framework

前面我们已经浏览学习了 Mesos APIs 及通过 APIs 如何编写代码等内容，现在让我们了解下如何在开发环境中运行你的代码。通常，你可以本地构建 Mesos 或者使用安装包安装，然后在你的机器上运行 Mesos master 和 slave。我更倾向于可以重复使用（可自由支配）的开发环境，这就是为什么我通常在 Vagrant 的环境中进行开发的原因。

10.4.1 在开发环境中部署

Mesosphere 团队有一个他们命名为 Playa Mesos 的开发环境。这个 Vagrant 环境安装了 Mesos、Marathon、Chronos 和 Docker，幸运的是，还提供了 Python 的 Mesos 本地库。

搭建 Vagant 环境

首先，如果没有安装 Vagrant 和 VirtualBox，那么你需要安装它们。这两个软件项目都是开源的，可以通过如下链接下载：

- www.vagrantup.com
- www.virtualbox.org

接着，通过运行以下命令克隆 Playa Mesos GitHub 库：

```
$ git clone https://github.com/mesosphere/playa-mesos
```

默认情况下，Playa Mesos 试图从 Mesosphere 库下载安装最新版本的 Mesos。对于本书来说（包括 Mesos 0.22.2），可能不需要安装最新版本。为了确保搭建 Vagrant box 时，已经安装了 Mesos 0.22.2，请添加如下一行来修改 config.json 的哈希值：

```
"mesos_release": "0.22.2-0.2.62.ubuntu1404"
```

最后，运行如下命令启动机器：

```
$ vagrant up --provision
```

测试最小的 framework

配备过程需要一些时间来下载和安装所需的安装包，所以请耐心地等待。在机

器完成配备之后，你就能够通过访问网址 http:// 10.141.141.10:5050 进入 Mesos 界面。如果一切都没有问题的话，就让我们继续下面的学习吧。

通过 SSH 切到运行的 Vagrant box 中并克隆本书的 Git 库：

```
$ vagrant ssh
vagrant@mesos$ git clone https://github.com/rji/
    ➥ mesos-in-action-code-samples
vagrant@mesos$ cd mesos-in-action-code-samples/chapter10
```

现在让我们继续，启动调度器：

```
vagrant@mesos$ ./scheduler-minimal.py zk://localhost:2181/mesos
```

观察输出

如果一切顺利的话，调度器应该已经注册到 Mesos master 中，并可以接受一个或多个资源提供，然后启动一个或多个任务。如果查看 Mesos 的网页管理界面，应该看到你的 framework 已注册并分配到一些资源，如图 10.4 所示。

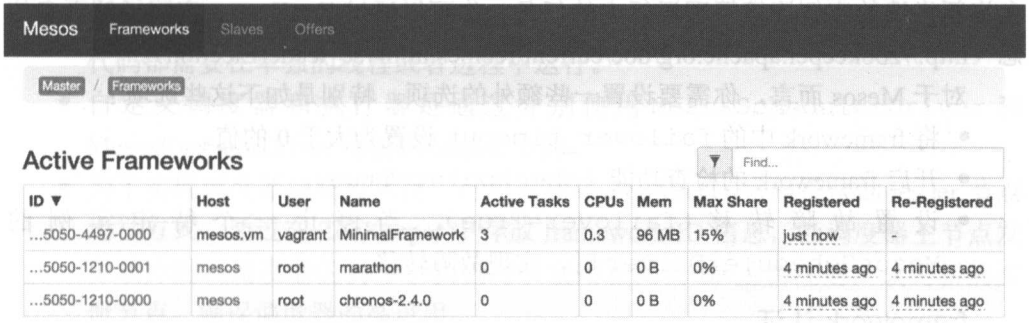


图 10.4 MinimalFramework 已经注册在 Mesos master 中，并且有三个任务正在运行

如果你点开某个任务的沙盒（sandbox），你将会看到类似如下的输出：

```
Hello from task 2c863b8a-1290-4849-958f-a3f2261e184a!
Hello from task d5aa1d8d-ea07-45f9-a703-d0e57ff88a22!
Hello from task aedc6839-4d44-47ec-8a23-fc7a20f1cd0b!
```

你会注意到，几个任务的输出都在执行器的标准输出上。因为你的 framework 只在一个节点上运行，所以 Mesos 创建了一个自定义执行器实例，所有任务都在这个执行器上运行。在我们的例子中，因为代码是一个简单的打印（`task.data`）程序，可以看到每个任务的 UUID 作为执行器实例的标准输出。

10.4.2 生产环境部署的注意事项

虽然本章只是framework开发的入门介绍，我还是想提几点作为结束语，比如你可以做些什么来添加一些生产级别的功能特性到你的framework中。总之，在这里我提出几点建议，分别是使调度器成为高可用，以及如何认证已启用认证功能的Mesos集群。但正如我前面提到的，Mesos frameworks是一个真正的分布式系统，所以一章远远不够涵盖开发一个可用于生产环境的framework的全部内容。

高可用性

虽然Mesos提供了一种使用相同的framework ID将framework注册到master中的方法，但这并不意味着这样部署的framework是高可用的；在调度器驱动程序注册到master之前，需要把这个逻辑需求嵌入到自己的framework中。

一种常见的做法是，通过ZooKeeper选举出一个调度器实例作为领导者，确保只有一个调度器实例注册到Mesos中。所注册的实例在ZooKeeper中保存它的framework ID，如果它出错了，剩下的实例就可以使用相同的framework ID进行注册，允许新当选的实例连接管理运行中的任务。你可以通过ZooKeeper官网了解更多信息（http://zookeeper.apache.org/doc/current/recipes.html#sc_leaderElection）。

对于Mesos而言，你需要设置一些额外的选项。特别是如下这些选项：

- 将framework中的`failover_timeout`设置为大于0的值。
- 开启framework的检查功能（`checkpoint=True`）。
- 设置故障转移（`failover=True`），当停止一个特定实例的`MesosSchedulerDriver`时，实现故障转移。

framework认证

本书第6章提到的*framework*认证是*Mesos framework*的可选功能。当应用framework注册到Mesos master节点时，framework认证功能允许你定义一种framework身份和密码注册到Mesos master中，并给系统管理员提供了一种控制frameworks注册到特定集群的方式。

```
message Credential {
  required string principal
  optional bytes secret
}
```

如果集群开启framework认证功能，你首先需要创建一个`Credential()`对象，然后把它作为参数传递给`MesosSchedulerDriver`，如下所示：

```
...
credential = mesos_pb2.Credential()
credential.principal = os.getenv('EXAMPLE_PRINCIPAL')
```

```
credential.secret = os.getenv('EXAMPLE_SECRET')
...
driver = MesosSchedulerDriver(
    ExampleScheduler(), framework, master, credential)
driver.run()
```

10.5 小结

本章中我们主要学习了如何开发一个 Mesos framework。我们介绍了类似如下主题：为什么需要编写自己的 framework 以及调度器和执行器的 APIs。请记住以下提到的几点：

- Mesos framework 是由调度器和执行器组成的。调度器和执行器分别可以通过实例化子类 mesos.interface.Scheduler 和 mesos.interface.Executor 来实现。
- 在调度器程序中，你只需要重写 `resourceOffers()` 这个方法。因为调度逻辑大部分都是在这个方法中进行处理的。
- 在执行器程序中，你只需要重写 `launchTask()` 这个方法。该方法中任何代码都需要在单独的线程或者进程中运行。
- 自定义调度器和执行器是通过分别使用 `MesosSchedulerDriver` 和 MesosExecutorDriver 连接到 Mesos 中的。
- 为了实现调度器的高可用性，可考虑使用 ZooKeeper 官网提供的 leader 节点选举方式。通过在 ZooKeeper 中存放 framework ID 信息，当调度器主节点发生故障时，调度器的另一个实例可以使用相同的 ID 重新注册，成为新的注册节点，确保调度器的高可用。

如果你正在寻找用其他语言开发调度器和执行器的例子，如 C++, Go, Haskell, Java 和 Scala，你可以从如下地址中获取 RENDLER 示例 framework：https://github.com/mesosphere/rendler。Mesos 项目还维护了一些基本的 framework 开发文档，可以在如下地址中查看：http://mesos.apache.org/documentation/latest/app-framework-development-guide.。

到此我们已经完成本书的学习！感谢大家抽空阅读这本书。在此我希望这本书能帮助你更好地理解 Mesos 的体系结构，并将其部署在你的数据中心中，由 Mesos 来管理运行应用程序和 Cron 任务。在附录 B 中列出了本书编写时出现的 Mesos frameworks 和相关的工具，包括适用 framework 的编程语言。

附录A 案例研究：Mesosphere DCOS，企业版Mesos分布式集群

在本书中，你已经了解了开源的 Apache Mesos 项目以及它如何能够在数据中心大型的机器集群中进行细粒度资源调度。同时你也知道了各种 Mesos 的应用案例，包括大规模数据处理作业的执行、应用程序和容器的部署以及计划任务的运行。但到了此时，你可能会对这么一个系统内的大量可移动组件感到不知所措，并关注寻找方法来确保某些组件的版本与其他特定版本的兼容性。在本书的第 3 部分已经提出了关于数据中心操作入门的思路和框架：应用程序管理、负载均衡、安全和访问控制，甚至包括对分布式系统开发的介绍。

本附录提供了一个数据中心操作系统（DCOS）上的应用案例：基于 Mesosphere 构建的 Mesos，一个创新的、企业级的，分布式操作系统。我介绍的主题包括 DCOS 如何为各种规模的企业提供全方位的 Mesos 解决方案；DCOS 如何处理分布式系统的包管理；以及如何使用 DCOS 上的 Jenkins 为部署自己的应用程序和容器开发一个可持续交付的途径，从而减少应用变更投入生产的时间。

A.1 DCOS 介绍

在过去的几年里，虽然企业组织似乎倾向于接纳开源技术，同时本书也为你在企业中部署 Mesos 集群提供指导，但一些企业还是对购买附带软件支持合同的全方

位解决方案比较感兴趣。其他公司正在接受开源技术，但当企业解决方案有显著的附加价值时，他们也不介意采用。通过 DCOS，Mesosphere 在开源的 Mesos 项目之上构建企业级软件，以帮助企业规模化它们的基础架构和自动化它们的应用部署，同时还为用户提供了舒适性和 24/7 全天候的客户支持、全面的测试组件，以及为用户和管理员提供的便捷工具。

DCOS 结合一些我在本书中提到的开源项目和商业组件来构建适用于内部部署和云设备的、易于管理和部署的 Mesos 集群，从而允许你快速地部署应用程序和容器。在 DCOS 中，为了确保平台的稳定，这些组件都全部一起测试，而且 Mesosphere 的工程师为每个组件编写了适用生产环境的配置。

注意：本附录介绍了在本书写作时的稳定版本 DCOS 1.4。

Mesosphere 以两种方式提供 DCOS 服务：

- 社区版（*CE*）——运行在亚马逊网络服务（AWS）上的免费版 DCOS，它也将扩展到其他供应商，包括微软的 Azure 和谷歌的云平台。
- 企业版（*EE*）——运行在内部部署或在云服务中的企业版 DCOS，按每个节点的基础计费。DCOS EE 包括理想的附加特性，如 24 / 7 / 365 全年的客服支持、Kerberos 认证、自定义安装和配置，甚至紧急补丁（如果需要的话）。

在进入本书使用 DCOS 的 Jenkins 进行持续交付的专题例子之前，让我们先了解一下 DCOS 架构的细节。读完这本书，在大多数情况下，就应该对其中的概念和组件比较熟悉了。在下一节中我会结合 DCOS 或某些情况下的 Jenkins 的背景，简要回顾一下每个组件的内容。

A.1.1 理解 DCOS 架构

在开发和运维 DCOS 的过程中，Mesosphere 通过类似于我在第 3 章第一次提到的方式配置 Mesos，用单一的 master 或三个 master 来确保高可用性。除了 Mesos master 守护进程，DCOS masters 还运行了在本节后面会介绍到的管理服务。根据你的工作负载的大小，再添加适当数量的 DCOS 代理到 masters 中。接下来和开源的 Mesos 部署不同的是 DCOS 还包括一个或多个、使用 Mesos 角色 `slave_public` 配置的公共节点。

公共节点的作用是为你提供运行反向代理、负载均衡器和专用节点（或节点集）上的其他面向外部服务的一种方法，允许在你的 DMZ 区域设置一个公共 DCOS 节点为用户提供流量服务，而集群中其余的运行应用程序的节点不用暴露给外部访问。你可以通过观察图 A.1 来查看这种部署场景。

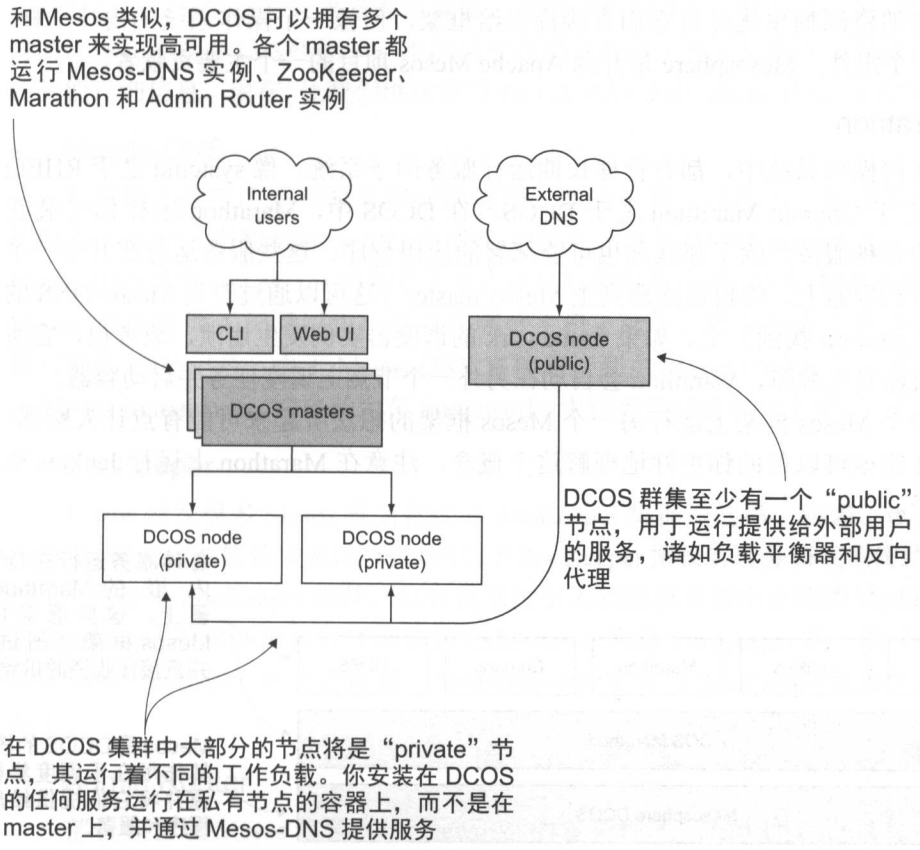


图 A.1 DOCS 部署的高级示意图

由于公共节点使用 `slave_public` 角色发送资源供给到 DCOS master，在这个 DMZ 节点上运行应用程序，你唯一需要做的事情就是把下面的字段添加到应用程序的 marathon.json 中：

```
"acceptedResourceRoles": [ "slave_public" ]
```

虽然 DCOS 有着比我在图中描述的还要多的组件，但你应该对 Mesos 如何使用 ZooKeeper 和 Mesos-DNS 如何通过 DNS 发布任务信息等过程比较熟悉了，所以为了清晰起见，我特意移除图中这些组件。但接下来的几节将提供 DCOS 中各个主要组件—Mesos，Marathon，Mesos-DNS，ZooKeeper 和称为 Admin Router 的反向代理如何在 DCOS 环境下运行的更多介绍。

Mesos

Mesos 是 DCOS 的核心，是在本书中所介绍的分布式系统内核。 Mesos 负责将

不同机器的资源抽象化并将它们直接提供给框架，但像一个操作系统的内核一样，它只是一个组件。Mesosphere 是开源 Apache Mesos 项目的一个主要贡献者。

Marathon

在任何操作系统中，都有管理长期运行服务的子系统。像 systemd 之于 RHEL; Upstart 之于 Ubuntu; Marathon 之于 DCOS。在 DCOS 中，Marathon 运行你安装在集群上的各种服务，除了那些你也可能部署的应用程序。这些服务运行在其中一个私有节点的容器上，将自己注册到主 Mesos master（这可以通过查询 Mesos-DNS 的 `leader.mesos` 找到）上。如果 framework 的调度器实例发生崩溃，或者说，它所运行的机器发生故障，Marathon 会自动在另外一个节点上调度任务并启动容器。

在一个 Mesos 框架上运行另一个 Mesos 框架的想法听起来可能有点让人疑惑，但图 A.2 应该可以帮助你更好地理解这个概念。注意在 Marathon 上运行 Jenkins 框架的例子。

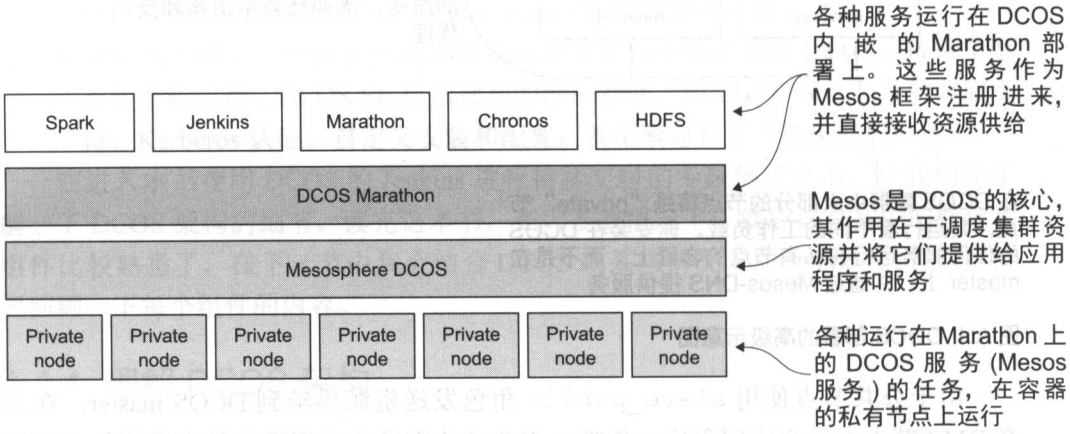


图 A.2 一个运行 DCOS 服务（Mesos 框架）的内置 Marathon 实例。Marathon 的各个实例可以在 DCOS Marathon 之上运行，为不同的团队提供部署应用程序的专有使用专用实例

对于在 DCOS 上部署 Jenkins，通常有如下的几件事情。

1. 一个新的 Marathon 应用程序被创建来运行 Jenkins master。在这个例子中，我们假设 Jenkins master 将运行在 Docker 容器中。
2. Marathon 在其中的一个 DCOS agents 上启动 Jenkins master 的一个实例。
3. Jenkins 启动并运行后，Mesos 插件注册到底层 Mesos master 中，就好像它是在你基础架构里某个地方的一台专用服务器上运行的。
4. Jenkins 注册后，就可以接受资源供给，并在 DCOS agent 上启动任务。

现在，我知道这看起来是有点像高层次的应用情景；它是包含在 Marathon 范围内的。现在不要担心这些细节。本附录中我稍后会提供有关如何安装和运行 Jenkins 的其他详细信息。现在，让我们继续学习 DCOS 如何使用 Mesos-DNS 处理服务发现。

Mesos-DNS

我在第 7 章首次提到服务发现的问题，即应用程序可能不一定需要知道依赖的服务（诸如一个数据库）运行在哪个主机上。每个 DCOS master 上运行着一个 Mesos-DNS 实例，向 Mesos 通知有关其正在运行的任务信息，并通过 DNS 发布信息。Mesos-DNS 提供 A 记录以识别任务在哪些主机上运行，以及包含主机 IP、协议和端口号的 SRV 记录。由于 Mesos-DNS 直接连接到主 Mesos master 下，而不是特定的 Mesos framework，它自动为在 frameworks 中运行的任务提供 DNS 记录。

ZooKeeper

Mesos 以及很多 Mesos 框架都依靠 ZooKeeper 集群做选举、协调，以及保持状态信息。因为这些依赖关系，这可以归功于 ZooKeeper 擅长实现分布式系统协调服务，DCOS 也包含 ZooKeeper 组件，但它是通过引入我在第 6 章中介绍的 Netflix 所开发的 ZooKeeper 管理和配置管理器组件 Exhibitor 来构建 ZooKeeper 的。除了在 DOCS 运行你自己框架的一些潜在配置，你可能就不再需要与 ZooKeeper 集群经常进行交互了。

Admin Router

部署和管理 Mesos 集群和 frameworks 的复杂性之一在于跟踪记录每个组件运行在哪些主机上以及监听的是什么端口。正如第 6 章所提到的，我建议尽可能在 Mesos master 前面使用 HAProxy，这样你的监测系统就可以使用单个 DNS 名称与当前的 leader 进行通信。类似我在第 7 和第 8 章中给 Marathon 和 Chronos 建议的方法，Admin Router 是 Mesosphere 应对上面所提问题的解决方案。

Admin Router 负责充当各种 DCOS 服务的反向代理，但它不是使用端口提供服务，而是使用特定的 URI。当你安装 DCOS 的 Jenkins 服务时，如下 url 将提供服务 http://dcos.example.com/service/jenkins; 没有必要担心调度器运行在哪个端口上！大多数与 DCOS 进行的用户交互要么通过 Admin Router 进行，要么至少用它来收集有关集群的信息（例如，SSH 登录到一个指定的节点）。通过这种方式，你可以限制暴露在集群网络之外的服务数量。

提示：Admin Router 的代码是开源的，可以在如下 url 中查看 https://github.com/mesosphere/adminrouter-public。

虽然我已经介绍了 Mesosphere 如何结合这些不同的组件去做一个稳定和鲁棒的

Mesos 部署，当你能明确地定义管理员和用户与系统的交互点时，真正的价值才开始显现出来。

A.1.2 与 DOCS 交互

当你花时间考虑使用什么样的基础组件与传统操作系统，如红帽企业版 Linux 或者 Ubuntu，进行交互的时候，组件基本上可分成以下三类：

- 包管理——包格式（rpm, deb）、包管理器（yum, apt）和基础仓库集（base, main）。你也可以选择启用来自供应商和第三方的实验性或测试的仓库。
- 命令行接口——当用户登录的时候，一个用于与系统交互的 shell（bash, sh, zsh）将会被启动。
- 用户图形接口——一个可选的图形用户接口，用于监控及管理系统。

在 DCOS 中，这些操作系统组件仍然存在，但在不同的抽象层中。接下来让我们来更加详细地了解它们在分布式系统例如 Mesos 上是如何工作的。

软件包管理

在写本书时，Mesosphere 为 DCOS 提供了两个包仓库，命名为 Universe 和 Multiverse。这些仓库主机分别存放生产包和实验包。在这些仓库中的一个包的元数据是可通过 DCOS CLI 处理并由 Marathon REST API 解析的一个 JSON 对象，这和我们在第 7 章学到的 Marathon 应用程序定义没有什么不同。

一些可供 DCOS 运行的服务如 Cassandra、HDFS 和 Kubernetes 等，需要大量的工作才能进行有效的部署。Mesosphere 团队使用这些软件包仓库，提供和维护全方位的解决方案以完全自动化的、容错的方式在你自己的数据中心部署这些服务。

Universe 仓库的文档已经很好地介绍了它的 schema，所以我不会在这里讨论。但是，如果你有兴趣为 DCOS 创建自己的包或想更详细地了解该 schema，请在 http://mesosphere.github.io/universe 查看中文档。

命令行接口

DCOS CLI 可以安装在你的笔记本电脑或工作站中，并与 DCOS 中各种服务进行交互。它为 DCOS 集群提供管理包、服务和节点的功能。DCOS 服务能够安装 DCOS 子命令，但这里包含了一些值得关注的内置的子命令：

- `config` ——获得并为 DCOS CLI 设置配置选项。
- `package` ——安装、管理、更新和卸载 DCOS 包。
- `node` ——列出和 SSH 进入属于 DCOS 集群中的节点。
- `marathon` ——部署和管理 Marathon 应用程序。

例如，在 DCOS 上使用 CLI 安装 Jenkins 包，你可以执行如下命令：

```
$ dcos package install --yes jenkins
```

就是这样！没有提供额外的基础设施，也不需要知道 Jenkins 运行在集群的哪个地方。软件包的维护者为你做了一切。可以运行命令 `dcos help <subcommand>` 来了解给定子命令的所有用法。

用户图形接口

虽然 DCOS CLI 可以完全使用命令行的方式来管理操作系统，但 Web 接口提供集群的相关信息，包括已安装的服务、运行中的任务以及属于集群的节点。图 A.3 显示 DCOS UI 主仪表面板，包括集群的 CPU 和内存分配情况、任务失败率、服务健康状态、运行中的任务和连接节点的数量。

在仪表板左侧的附加选项卡中，允许你查看集群上的服务资源分配情况，以及单个节点的资源利用率和健康状况。

提示：如果你想启动自己的 DCOS 集群，可以在 mesosphere.com/product 中找到更多的关于如何操作的信息。

关于 DCOS 的一个好处就是，它给你在 Mesos 上附加企业特性的功能。读完本书，你不仅学会了如何部署和使用 Mesos，在某种程度上，也在这个过程中学到了很多关于 DCOS 的知识，或许你还没有意识到这一点。

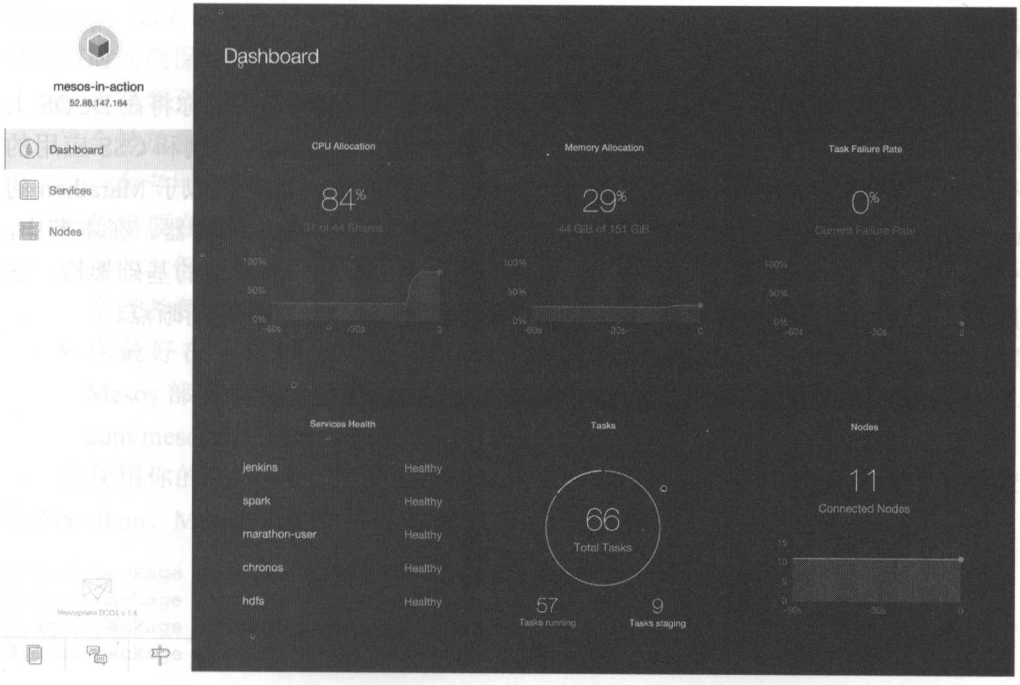


图 A.3 DCOS Web 接口展示

现在，本书已经接近尾声，让我们开始最后一课的学习，你将在本课中学习到：使用 Jenkins 与 DCOS（和 Mesos 与 Marathon，同样如此！）去不断测试和部署应用程序的更改，以便你和你的工程团队可以快速并轻易地开发新功能和修复 bug 投入生产环境。

A.2 使用Jenkins和Marathon执行持续部署

在全书中，我一直讲述的主题是 Mesos 怎样通过抽象多台机器资源为单一的实体来使你简化操作。我已经讲述了如 Marathon 和 Aurora 这样的应用平台和一些你可以试验这些项目的示例。但是到目前为止，你还不能在一个紧密结合的端到端的例子中应用多个章节中的概念。

我会提供一个包含目前你所学习到的知识的用例。在此附录的最后，你将摸索如何使用 Jenkins 这个热门的、开源的持续集成系统。连同 Mesos，Marathon 以及 DCOS 一起，它轮询着 Git 版本库代码的变更并且自动触发软件项目的 Docker 镜像的构建。如果构建成功地完成，新的 Docker 镜像将被推送到 Docker Hub 上，而新的应用版本将会部署到 Marathon 上。最终也会作用到 Mesos 上。你应用部署的烦恼将会自动减少，取而代之，你可以集中时间和精力在编写代码上。

如果上述有任何让你感兴趣的地方，那真是太好了！让我们一起开始吧。

A.2.1 为应用持续部署准备 DCOS

本节将基于前面你对 DCOS 的学习情况提供一个用例，在里面你将在 DCOS 上使用 Jenkins 来持续构建和部署本书补充材料中的一个 HTML 示例和 CSS 应用的变更。我也会为你介绍另外一个名为 Marathon-LB 的项目，它类似于 Marathon 的 servicerouter.py 脚本，可用来自动地配置和重载 HAProxy 负载均衡器。在本节中，你将通过运行少数的命令和编写 Jenkins 构建脚本来构建一个小型的基础架构，它用来提供应用服务和负载均衡用户访问。我会在图 A.4 上描述得更清晰点。

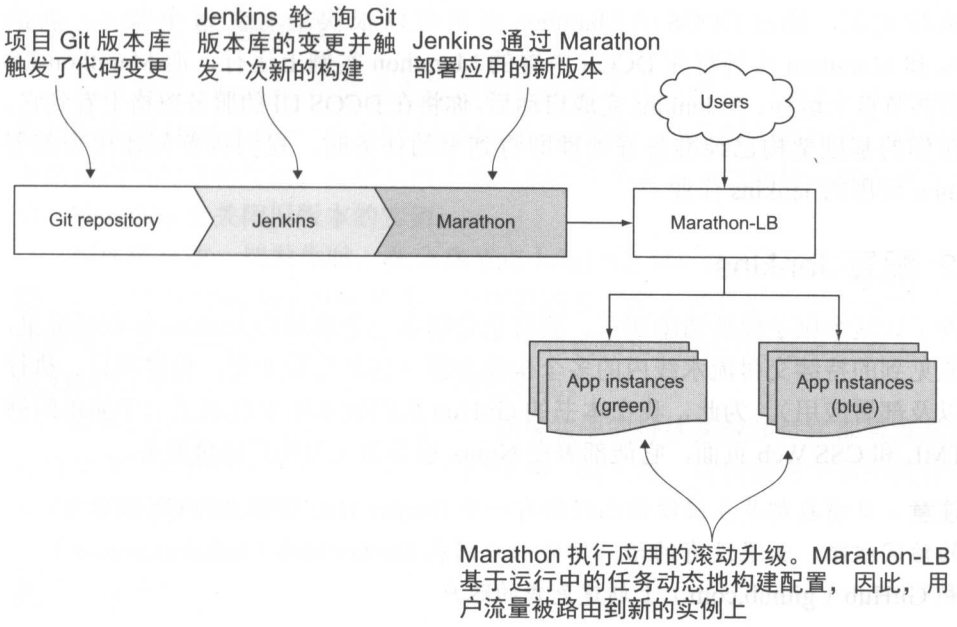


图 A.4 使用 Git、Jenkins、Marathon 和 Marathon-LB 持续部署流水线

我将会演示怎么通过 DCOS 集群执行这个实例来运行 Jenkins 和部署示例的 Web app。如果你只想使用你所学到的 Mesos 集群（可能是通过阅读了本书），你需要花点时间确保如下几点：

- 你需要至少一个 Marathon-LB 在其上运行的公共节点。通常来说，这个节点会被放置在 DMZ 区来接收公网（或者内网用户）的流量。因此，我推荐在一台专用的 Mesos slave 上配置 `–default_role=slave_pubic`。
- 你需要在 Mesos master 上安装 Marathon 和 Mesos-DNS。尽管这不是严格要求的，但你最好把部署 Marathon 当作运行其他 Mesos framework（也包括一个或者多个 Marathon 实例）的方法。
- 你最好在工作站上安装 DCOS CLI，请注意 DCOS CLI 也能用在开源的 Mesos 部署上。项目的 README 文件中有配置指引，请访问 https://github.com/mesosphere/dcos-cli。

在使用你的集群 URL 配置完 DCOS CLI 后，通过执行下面的命令使用它来安装 Marathon、Marathon-LB 和 Jenkins 安装包：

```
$ dcos package update
$ dcos package install --yes marathon
$ dcos package install --yes marathon-lb
$ dcos package install --yes jenkins
```

执行完后，通过DCOS的Marathon实例可以很快地看到每个服务。新的Jenkins和Marathon实例将在DCOS内置的Marathon实例上运行，而Marathon-LB将在公共节点上运行。在Jenkins完成启动后，你将在DCOS UI的服务窗格上看到它。

在你的基础架构已经准备好处理即将到来的任务时，我们接着创建用来部署Marathon应用的Jenkins作业。

A.2.2 配置Jenkins

为了让这个例子保持清晰明了，我将会介绍在一个单独的Jenkins作业环境里，通常能见到的持续交付流水线内的多个构建步骤（包括轮询变更、构建项目、执行测试以及部署应用）。为此，我在本书的GitHub代码版本库中收录了一个简单的静态HTML和CSS Web页面，它能部署在Nginx服务器上为用户提供服务。

> **注意**：目前我都是在假设你已经拥有一个Docker Hub的账户和Git版本库的访问权限。如果没有的话，你可以分别在Docker Hub（hub.docker.com）和GitHub（github.com）上创建免费的账户。

要在Jenkins中设置这个构建并部署的作业，你需要访问Jenkins的Web接口。在DCOS中，你能从首页Services下面的Dashborad主页来访问Jenkins。而在Mesos中，你能够通过浏览Frameworks选项卡并点击framework的Web URL链接来访问Jenkins。

在访问Jenkins Web接口后，执行下面的步骤。

1. 创建一个新的Jenkins作业并为之命名。在本例中，让我们假设该作业叫做“demo-app-build-and-deploy”。
2. 在作业的配置选项页面，配置Git插件来轮询你的应用代码版本库的变更。（请随意复制本书GitHub版本库上的这个附录示例代码到你自己的GitHub库里，并进行测试。）
3. 创建一个新的构建步骤来执行shell脚本。在这，你可以写一个脚本来创建Docker镜像，并推送它到Docker Hub上，然后再使用新的Docker镜像标签更新Marathon应用。我已经在Git版本库中存放了一个示例，就个人而言，我更喜欢把这个构建脚本放到我的应用代码旁边的版本控制里。
4. 可选：配置Jenkins在作业开始和完成时发送电子邮件通知或者聊天消息通知。这样，你和团队中的成员就都能知晓有个变更已经被部署了。

在创建、使用了你的Git版本库的URL，轮询调度和已构建脚本的Jenkins作业后，你既可以等待作业运行，又可以手动触发一个构建。

A.2.3 持续部署实战

无须多想，Jenkins 作业会克隆你的代码版本库，创建一个新的 Docker 镜像，把它推送到 Docker Hub 上，然后向 Marathon 部署一个新版本的应用。Marathon 接着执行应用的滚动升级，让新旧两个版本的应用共存。一旦新版本的健康检查通过后，Marathon 会关闭旧版本的实例。

当这所有的一切发生时，在公共节点上运行的 Marathon-LB 被订阅到通过 API 路径可访问的 Marathon 时间流中。当应用的新实例上线且旧实例下线时，该负载均衡器会更新 HAProxy 配置来确保请求都能被路由到应用的健康实例上。在执行 DCOS CLI 上的几条命令后，这一切都能实现。

我明白这里的示例可能太过基础，而现实中的持续集成和持续部署流水线通常会更复杂，并且应用也需要引入不同的依赖。没关系，请根据这个示例进行满足你的需求和应用需求的构建。

A.3 小结

在本附录中，你学习了 Mesosphere 是怎样提供企业级别的 Mesos 发行版的，你也学习了 Jenkins、Marathon 和 Mesos 是怎样实现在无人干预的情况下自动部署应用变更的。下面是需要记住的几点。

- 在 Mesosphere DCOS 是构建在 Mesos 分布式系统内核之上的企业级的发行版。DCOS 为企业提供了规模化部署应用容器的全方位解决方案。
- 在 DCOS 部署中的节点分为两种类型：公共的和私有的。私有节点组成了集群的绝大部分，而少量的公共节点能运行对外的服务，如反向代理和负载均衡器。
- Mesosphere 工程师为 DCOS 维护越来越多的分布式服务，包括 Cassandra, Chronos, HDFS, Jenkins, Kafka, Marathon, Spark 以及其他。实际上就是任何 Mesos framework 都能在 DCOS 上运行。
- 通过在 DCOS 上安装 Jenkins 和 Marathon 服务，你能够对你的应用进行自动化地持续构建、测试以及部署。Jenkins framework 可按需动态启动或销毁基于容器构建的 slave。
- 通过使用 Jenkins 和 Marathon 来部署应用，在持续集成流水线成功执行时，你能够持续地部署应用的升级。Marathon-LB 订阅 Marathon 的事件流能自动更新负载均衡器的配置文件并且重载服务。

附录 B 提供最新的 Mesos frameworks 列表，每个 framework 都能被部署在你的一个 Mesos 集群上，而且有些对 DCOS 是可用的，感谢 Mesosphere 工程团队的努力成果。

附录B Mesos框架与工具的列表

Mesos 被设计用来支持在一个计算机集群上运行多个框架，以提高整体资源利用率。各社区中围绕 Mesos 运行的各种应用程序做了不少工作。

本附录中，我列出一些知名的和自发布以来有积极维护的 Mesos 框架。我也介绍 Mesos 框架开发所支持的编程语言，允许你除了使用 C++, Java, Scala 和 Python 之外的语言来开发自己的框架。最后，我会介绍社区中可以用于配置管理、监控、服务发现和负载均衡等工具。

B.1 Mesos 框架

本书出版时，有几个开源的 Mesos 框架可以使用。有些是针对特定目的构建的——如 Aurora、Chronos 和 Marathon，而有一些是在 Mesos 模型上运行良好的现有分布式服务，其中包括（但不限于）Cassandra、Jenkins 和 Spark。本节会介绍你可以直接用来构建你的集群的 Mesos 框架。

B.1.1 应用管理和批调度

Mesos 的一个主要用途就是部署长期运行的应用到 Mesos 集群上，有效地利用 Mesos 来分发和运行容器。本节介绍可用于在 Mesos 集群上部署应用程序和批处理

作业的框架，类似于第 7 到 9 章讨论的话题。

名字	描 述	更多信息
Aurora	Apache Aurora 是 Twitter 为了管理长期运行的服务和调度任务而开发的框架	https://aurora.apache.org
Chronos	Chronos 是一个由 Airbnb 开发的以容错方式运行调度数据处理作业的框架。它支持以 ISO 8601 为基础的计划和作业依赖	http://mesos.github.io/chronos
Cloud Foundry	Huawei 开发了一个在 Mesos 集群上运行开源的 Cloud Foundry 的 PaaS 服务的 Mesos 框架	https://github.com/mesos/cloudfoundry-mesos
Docker Swarm	Swarm，由 Docker 团队开发的容器聚类分析和业务流程的工具，可被配置来使用 Mesos 集群实现对计算资源的管理	https://github.com/docker/swarm/blob/v1.0.1/cluster/mesos/README.md
Jenkins	Jenkins 是一个开源的为软件开发和应用管理提供持续集成和部署的工具。通过使用 Mesos 插件，Jenkins 可以在 Mesos 集群上弹性伸缩其基础架构	https://github.com/jenkinsci/mesos-plugin
Kubernetes	Mesosphere 团队为谷歌开源的容器调度器开发了可以运行在 Mesos 集群上的框架。这允许 Kubernetes 与其他 Mesos 框架一起运行在 Mesos 集群上，如 Spark	https://github.com/mesosphere/kubernetes-mesos
Marathon	Marathon 是由 Mesosphere 开发和维护的开源 Mesos 框架。它可以在 Mesos 上部署应用和长期运行的服务	http://mesosphere.github.io/marathon
PaaSTA	PaaSTA 是一个由 Yelp 为了在 Mesos 上运行服务和调度作业而开发的平台。它是基于一些其他的开源项目建立的，包括 Marathon、Chronos 和 Docker	https://github.com/Yelp/paasta
Singularity	Singularity 是 HubSpot 为了在 Mesos 集群上启动长期运行的，有调度计划的或者一次性的任务而开发的框架	https://github.com/HubSpot/Singularity

B.1.2 数据处理

Mesos 的第一个用途，正如其原始研究论文中所提到的，是为了执行数据处理任务。事实上，Apach Spark 项目是由原来 Mesos 项目部分成员开发的，为了证明特定的数据处理框架比通用的框架更加有价值。本节介绍了一些流行的数据处理框架，并为每一个项目提供说明和项目的官网 URL。

名字	描 述	更多信息
Cook	Cook 是一个由 Two Sigma 开发的 Mesos 批处理调度器。它被设计来支持多个用户和抢占低优先级的任务，并能为运行 Spark 作业提供多租户环境	https://github.com/twosigma/cook

续表

名字	描　述	更多信息
DPark	DPark 是 Apache Spark 的 Python 克隆版本，包含了可以在集群上运行作业的内嵌组件支持	https://github.com/douban/dpark
Hadoop	Apache Hadoop 是一个流行的数据处理框架和生态系统。这是第一个移植到 Mesos 上的应用，被原创性的 Mesos 论文广泛引用	https://github.com/mesos/hadoop
Kafka	Apache Kafka 是一个分布式的、高吞吐量的发布 / 订阅（Pubsub）消息系统。通过在 Mesos 上运行 Kafka，像 Apache Spark 一样，你可以灵活伸缩 Kafka，与其他框架一起消耗流数据	https://github.com/mesos/kafka
Myriad	Apache Myriad 目前是 Apache 的孵化项目，让 Hadoop YARN（MapReduce V2）可以运行在 Mesos 集群上。通过在 Mesos 上运行 YARN，YARN 应用程序可以像其他 Mesos 框架一样，能够共享相同的物理基础架构	https://myriad.incubator.apache.org
Spark	Apache Spark 是一个流行的开源数据处理框架，是 Mesos 首个专门为数据处理而建立的框架。与 Hadoop 基于磁盘的 map/reduce 模式相比，Spark 可以将数据集加载到内存中，在某些情况下，能提升 10 倍于 Hadoop 的性能	https://spark.apache.org
Storm	Apache Storm 是一个专注于实时计算的开源的、流处理系统	https://github.com/mesos/storm

B.1.3 分布式数据库和存储

分布式数据库和文件系统有着自身的集群和复制机制，像 Cassandra 和 HDFS。通常这些服务将运行在专用的机器组上。因为 Mesos 提供了用于分布式计算的原语，已经出现一些在一个单一的通用 Mesos 集群上以完全自动化的方式运行的分布式系统的工作成果。本节提供了一个运行在 Mesos 上的分布式数据库和文件系统的列表。

名字	描　述	更多信息
ArangoDB	ArangoDB 是一个可以处理 JSON 文档，图表和键 / 值对的、开源的、分布式的 NoSQL 数据库	https://github.com/arangodb/arangodb-mesos
Cassandra	Apache Cassandra 是一个用于管理海量数据的可伸缩的 NoSQL 数据库。包括苹果、CERN 和 Netflix 等公司都将它应用到生产环境上	http://mesosphere.github.io/cassandra-mesos
Ceph	Ceph 是一个具有容错性和自愈能力的分布式文件系统。英特尔大数据分析团队创建了一个可以在 Mesos 上扩展 Ceph 集群的 Mesos 框架	https://github.com/Intel-bigdata/ceph-mesos

续表

Elasticsearch	Elasticsearch 是一个基于 Apache Lucene 的由 Elastic 开发的，开源的、分布式的搜索和分析服务器	https://github.com/mesos/elastic-search
Etcd	Etcd 是由 CoreOS 开发的分布式键 / 值存储数据库	https://github.com/mesosphere/etcd-mesos
HDFS	Hadoop 分布式文件系统（HDFS）是一个被设计用来运行在商用硬件上的分布式的，且具有容错性的文件系统	https://github.com/mesosphere/hdfs
MemSQL	MemSQL 是一个分布式的 SQL 内存数据库	https://github.com/memsql/memsql-mesos
Riak KV	Riak KV 是由 Basho 开发的一个强大的键 / 值存储数据库	https://github.com/basho-labs/riak-mesos

B.2 Mesos相关工具

相对于传统的数据中心架构，Mesos 提供了一种完全不同的方式，允许你为应用程序调度多台机器的资源，而不是单个机器上的资源。当你知道一个应用程序运行在单个主机上时，配置负载均衡器或 Web 应用程序获取数据库连接是比较容易的。但是当应用程序可以在成百上千的节点中的任何一个节点上运行时，事情就变得有些复杂了。

本节介绍了你在自己的环境中可以使用的一些工具。这些措施包括用于开发自己的 Mesos 框架的语言绑定、负载均衡和服务发现方案，监控和配置管理脚本和一些 Vagrant 开发环境。

B.2.1 语言绑定

Mesos 的语言绑定允许你选择语言编写 Mesos 框架。一些被我称为本地绑定（*native bindings*）的绑定是由 Mesos 本身维护和发布的，其他的绑定是由社区开发和维护的，这些都支持你或你的开发团队用自己熟悉的语言编写框架。

Mesos Native

正如我在第 10 章所提到的，Mesos 允许你灵活地使用 C++、Java、Scala 和 Python 等语言编写框架。这里有一些在线资源提供参考：

- http://mesos.apache.org/api/latest/c++
- http://mesos.apache.org/api/latest/java
- https://github.com/apache/mesos/blob/0.22.2/src/python/interface/src/mesos/

interface/__init__.py

此外，一定要阅读Mesos框架开发指南（位于http://mesos.apache.org/documentation/latest/app-framework-development-guide），和RENDLER例子（位于https://github.com/mesosphere/rendler）。

社区维护

Mesos社区的一些成员和组织创建并维护了语言绑定等工具，允许你在编写Mesos框架的时候有更多的编程语言可以选择：

- Clojure—https://github.com/dgrnbrg/clj-mesos
- Erlang—https://github.com/mdevilliers/erlang-mesos
- Haskell—https://github.com/iand675/hs-mesos
- Pure Java—https://github.com/groupon/jesos
- Go—https://github.com/mesos/mesos-go
- Perl—https://github.com/mark-5/perl-mesos
- Pure Python—https://github.com/wickman/pesos
- Rust—https://github.com/spacejam/mesos-rs

B.2.2 负载均衡和服务发现

因为你不知道一个特定的应用程序或服务正在哪台机器上运行，所以你需要一种方法利用已经运行在Mesos（或你的Mesos框架）上的任务信息来发现并连接到它们。

名字	描　述	更多信息
Aurproxy	Aurproxy是Apache Aurora的负载均衡器。当发生变化时，它为Nginx生成配置并优雅地重新加载服务	https://github.com/tellapart/aurproxy
Bamboo	Bamboo是Marathon的负载均衡器，它基于Marathon API提供的可用状态信息来配置HAProxy。它还包括一个用户界面和定义HAProxy ACLs的API	https://github.com/QubitProducts/bamboo
Marathon-LB	Marathon-LB是由Mesosphere为Marathon开发的负载均衡器。它可以通过Marathon API获取状态信息或订阅Marathon的事件流来动态构建HAProxy配置和优雅地重新加载服务	https://github.com/mesosphere/marathon-lb
Mesos-Consul	Mesos-Consul从Mesos中轮询发现所有框架中运行的任务信息并发布信息给Consul，然后就可以通过DNS和Consul HTTP API进行访问了	https://github.com/CiscoCloud/mesos-consul
Mesos-DNS	Mesos-DNS是一个无状态的DNS服务器，可以从Mesos中轮询发现所有框架中运行的任务信息。然后通过其内置的DNS服务器和HTTP API提供服务信息	https://mesosphere.github.io/mesos-dns

B.2.3 监控和管理

当你部署任何新的服务到你的环境上，特别是对于 Mesos，选择什么样的方式去监控服务的健康状态以及管理和部署服务配置的变更是很重要的。本节介绍可以用来监视你的 Mesos 集群健康状态的工具，以及眼下最为流行的 3 个配置管理工具插件。

监控

为了监控 Mesos 服务和框架，你通常可以使用任何已经安装的监控工具。例如，你可以使用 Elasticsearch、Logstash 和 Kibana（ELK）stack，或 Splunk 实现跨集群的集中化日志管理，你也可以使用 Nagios、Icinga，或第三方监测工具实现整体的监控和报警。这里有一些你可能会感兴趣的开源项目：

- Collectd plugin for Mesos—https://github.com/rayrod2030/collectd-mesos
- Exhibitor (a supervisor for ZooKeeper)—https://github.com/Netflix/exhibitor
- Nagios checks for Mesos—https://github.com/opentable/nagios-mesos
- Nagios checks for ZooKeeper—https://github.com/apache/zookeeper/tree/trunk/src/contrib/monitoring
- Prometheus exporter for Mesos—https://github.com/prometheus/mesos_exporter
- Satellite (monitoring service for Mesos)—https://github.com/twosigma/satellite

配置管理

在许多组织和环境中，配置管理工具使系统管理员能够以一对多的方式管理机器。在本书写作之时，主要有三种流行的配置管理工具：Ansible、Chef 和 Puppet。社区成员已编写了代码来调用这些工具对 Mesos 和 ZooKeeper 集群提供服务，这里我想强调一些较为知名的项目。

Ansible 是一个配置管理工具和业务流程引擎，允许你通过 SSH 对机器部署变更，而不需要在管理的机器上运行代理。下面的两个版本可以给 Mesos 与 ZooKeeper 的集群提供服务：

- https://github.com/AnsibleShipyard/ansible-mesos
- https://github.com/AnsibleShipyard/ansible-ZooKeeper

Chef 允许系统管理员使用领域特定语言，或 DSL 来描述它们架构所需的状态。如下两个 Chef 版本将允许你为 Mesos 和 ZooKeeper 集群提供服务：

- https://supermarket.chef.io/cookbooks/mesos
- https://supermarket.chef.io/cookbooks/zookeeper

Puppet 允许管理员通过使用 DSL 声明他们的架构所需的状态。如下两个 Puppet 模块允许你为 Mesos 和 ZooKeeper 集群网提供配置管理服务：

- https://forge.puppetlabs.com/deric/mesos
- https://forge.puppetlabs.com/deric/zookeeper

B.2.4 Vagrant 环境

Vagrant 环境可以使得在笔记本电脑上支持使用 Mesos 集群。如下两个软件还特别安装了 Mesos、Marathon 和 Docker：

- https://github.com/mesosphere/playa-mesos
- https://github.com/tayzlor/vagrant-puppet-mesosphere

译者简介

余何

昵称：众神的大师兄，湖南长沙人也；高效运维公众号专栏作者；著有《PaaS 实现与运维管理》一书，具有十余年数据中心运维管理经验；拥有国家软件设计师、PMP 项目管理认证、Juniper 互联网专员（JNCIS）和 NetApp 解决方案架构师（NCSA）的职业资格证书；热衷于开源技术，广结天下英豪，以运维心灵捕手著称。

陈秋浩

基础架构资深工程师，拥有 6 年大型 IT 数据中心运维和开发经验，早期活跃于基础架构服务交付和异常事件处理一线。爱好开发，拥抱开源技术，于 2014 年年末借 Docker 和 Mesos 技术兴起之势，辅以业界日益成熟的自动化运维理念，负责开发搭建企业内部基于Mesos+Marathon的高容错性、弹性伸缩的 Docker 平台。

杨永帮

2011 年毕业于中山大学软件工程专业，拥有多年的大型金融集团的 IT 基础架构运维的丰富经验，解决过 IT 基础架构的各种疑难杂症问题；深深感受到了云计算的发展带来的运维方式的变革，目前致力于 Mesos 与 Docker 的研究工作。

Broadview® www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Mesos实战

现代数据中心的环境非常复杂，尤其在你把Docker和其他基于容器的系统混入其中的时候，“简化数据中心”这个需求就变得更迫切了。Mesos是一个开源的集群管理平台，它能把整个数据中心转化为一个独立的资源池,让你像在使用一台超级计算机一样，对上面的计算、内存及存储资源进行分配、自动化操作及伸缩操作。

《Mesos实战》一书为读者介绍Apache Mesos集群管理器及以应用程序为中心的基础架构概念。本书充满了有用的数据图表及实践指导，它将指引你迈出创建一个高可用的Mesos集群的第一步，接着在生产环境中部署应用程序，最后编写适合自己数据中心的“本地”Mesos framework（计算框架）。你将学习到如何对若干个节点进行弹性伸缩，同时通过Linux和Docker容器保证不同的进程间能实现资源隔离。你也将学习到如何使用热门主流的framework来部署应用程序的实践技术。

本书包括

- 搭建启动你的第一个Mesos集群
- Mesos的调度、资源管理及日志记录
- 使用Marathon、Chronos和Aurora部署容器化的应用程序
- 使用Python编写Mesos framework

读者需要熟悉数据中心管理的核心理念，也需要了解Python或者类似编程语言的基础知识。

“你难以找到一个比Roger更好的导师，或者一本比《Mesos实战》更好的书。”

——《序》Florian Leibert,Mesosphere

“本书可以为你在部署Apache Mesos时指明最佳实践，并帮你避开隐藏的陷阱。”

——Marco Massenzio, Apple

“Ignazio精通他的业务，更重要的是他知道如何讲解它。”

——Morgan Nelson, getaroom.com

“通过阅读本书，你不仅能学习Mesos，还能学到整个生态系统。”

——Thomas Peklak, Emakina CEE

MANNING

策划编辑：符隆美
责任编辑：徐津平
封面设计：王 乐

上架建议：容器

ISBN 978-7-121-31164-2

9 787121 311642

定价：69.00元